USENIX Association

Proceedings of USENIX ATC '16

2016 USENIX Annual Technical Conference

June 22–24, 2016
Denver, CO, USA

Conference Organizers

Program Co-Chairs

Ajay Gulati, *Zerostack, Inc.*

Hakim Weatherspoon, *Cornell University*

Program Committee

Mohit Aron, *Cohesity*

Mahesh Balakrishnan, *Yale University*

Theophilus Benson, *Duke University*

Randal Burns, *John Hopkins University*

Haibo Chen, *Shanghai Jiao Tong University*

Byung-Gon Chun, *Seoul National University (SNU)*

Paolo Costa, *Microsoft Research*

Dilma Da Silva, *Texas A&M University*

Angela Demke Brown, *University of Toronto*

Fred Douglis, *EMC*

Rodrigo Fonseca, *Brown University*

K. Gopinath, *Indian Institute of Science (IISc)*

Haryadi Gunawi, *University of Chicago*

Indranil Gupta, *University of Illinois at Urbana–Champaign*

Andreas Haeberlen, *University of Pennsylvania*

Tim Harris, *Oracle*

Anne M. Holler, *FUEGO*

Jon Howell, *Google*

Hani Jamjoom, *IBM*

Anthony Joseph, *University of California, Berkeley*

Geoff Kuenning, *Harvey Mudd College*

Peter Pietzuch, *Imperial College London*

Sriram Rao, *Microsoft*

Benjamin Reed, *Facebook*

Scott Rixner, *Rice University*

Henry Robinson, *Cloudera*

Leonid Ryzhyk, *Samsung Research America*

Liuba Shrira, *Brandeis University*

Nisha Talagala, *Parallel Machines*

Theodore Ts'o, *Google*

Dan Tsafrir, *Technion—Israel Institute of Technology*

Andy Tucker, *Bracket Computing*

Zhen Xiao, *Peking University*

Noa Zilberman, *University of Cambridge*

External Reviewers

Masoud Saeida Ardekani

Riccardo Bettati

Carlo Curino

Chris Douglas

Steven Eliuk

Raman Grover

Andrew Hilton

Alekh Jindal

Konstantinos Karanasos

Ricardo Kollerr

Cheng Li

Radia Perlman

Rahul Potharaju

Michael Rabinovich

Rayman Preet Singh

Weijia Song

Rachel Traylor

Carl Waldspurger

Markus Weimer

Kai Zeng

USENIX ATC '16
2016 USENIX Annual Technical Conference
June 22–24, 2016
Denver, CO, USA

Wednesday, June 22, 2016

Datacenter Networking

File and Key-Value Systems

(Wednesday, June 22 continues on the next page)

Mobile and Apps

Systems and Network Security

Thursday, June 23, 2016

Cloud, Coordination, and Consensus

Architectural Interaction

Caching and Indexing

Energy vs. Performance

Network Design and Usage Studies

Friday, June 24, 2016

Data Is Now Big Data

Virtualization

Operating Systems

Message from the USENIX ATC '16 Program Co-Chairs

Welcome to the 2016 USENIX Annual Technical Conference.

This year's program committee has put together a program of 47 refereed papers and four practitioner talks. These papers and talks span a wide range of topics, covering both novel research contributions and practical ideas in storage systems, networking, memory management, data analytics, parallel and distributed systems, mobile computing and consumer electronics, security, reliability, and virtualization.

For the traditional refereed papers track, we received a near-record number of paper registrations and submissions this year. Authors registered 297 abstracts, of which 248 were submitted as complete papers. The program co-chairs rejected 11 papers up front due to serious format violations. Of the submitted papers, 35 were short papers, which had to be at most 5 pages long plus references, and the other 213 were full-length papers, which had to be at most 11 pages long plus references.

Reviewing was single-blind, done by the program committee in two rounds, with a few external reviews. In the first round, each of the 248 submitted papers received two reviews. Papers receiving at least one "weak accept" or better (i.e., "accept" or "strong accept") review moved on to the second round. In total, 166 papers moved on, and 82 papers were tentatively rejected. In the second round, each paper received at least two more reviews. After round two, 61 papers were tentatively rejected. Next, we discussed the remaining papers online; a total of 105 papers were discussed online after round two.

Altogether, more than 820 reviews were completed.

After two phases of review, an online discussion was conducted among reviewers, during which the program committee decided to accept 17 highly-ranked papers, reject 16 more papers, and to further discuss 72 papers during the in-person program committee meeting. The meeting was held on 7th and 8th April at the Facebook campus in Menlo Park, California. Four PC members called in; one of them could not attend; the other members all attended in person. Over a period of one and half days, the committee decided to accept 47 papers, including the 17 papers that were accepted earlier. Among these 47 papers, one was a short paper.

We continued the track for industrial practitioners talks and accepted five talks. We also continued the tradition of inviting best of the rest talks from the best papers at other USENIX-sponsored conferences and invited talks from FAST, NSDI, LISA, and SOSP.

The program committee was comprised of 36 members, including the two co-chairs. Fifteen of them were affiliated with industrial organizations and 21 were affiliated with academic organizations. The committee represented three continents and seven countries. Program committee members were allowed to submit papers. One of the co-chairs also submitted one paper and was conflicted on a few of them. All of the co-chairs' papers and conflicts were handled using a token-based review process to guarantee complete anonymity. We followed conventional rules for handling conflicts of interest: conflicted members (or co-chairs) left the room during discussion of conflicted papers.

In addition to the authors that submitted their work for consideration, the program committee, and the external reviewers, we would like to thank the USENIX staff that took care of all organizational details. Their help made our jobs a lot easier and allowed us to focus on reviewing papers and putting together the technical program. We would also like to thank Facebook for their generosity in hosting the PC meeting.

We hope that you enjoy the conference, and thank you for participating in the USENIX ATC community.

USENIX ATC '16 Program Co-Chairs
Hakim Weatherspoon, *Cornell University*
Ajay Gulati, *Zerostack Inc.*

FLICK: Developing and Running Application-Specific Network Services

Abdul Alim[†], *Richard G. Clegg*[†], *Luo Mai*[†], *Lukas Rupprecht*[†], *Eric Seckler*[†], *Paolo Costa*[♯†], *Peter Pietzuch*[†], *Alexander L. Wolf*[†], *Nik Sultana*[*], *Jon Crowcroft*[*], *Anil Madhavapeddy*[*], *Andrew W. Moore*[*], *Richard Mortier*[*], *Masoud Koleini*[♭], *Luis Oviedo*[♭], *Derek McAuley*[♭], *Matteo Migliavacca*[‡]

[†]Imperial College London, [♯]Microsoft Research, [*]University of Cambridge, [♭]University of Nottingham, [‡]University of Kent

Abstract

Data centre networks are increasingly programmable, with *application-specific* network services proliferating, from custom load-balancers to middleboxes providing caching and aggregation. Developers must currently implement these services using traditional low-level APIs, which neither support natural operations on application data nor provide efficient performance isolation.

We describe FLICK, a framework for the programming and execution of application-specific network services on multi-core CPUs. Developers write network services in the FLICK *language*, which offers high-level processing constructs and application-relevant data types. FLICK programs are translated automatically to efficient, parallel *task graphs*, implemented in C++ on top of a user-space TCP stack. Task graphs have bounded resource usage at runtime, which means that the graphs of multiple services can execute concurrently without interference using cooperative scheduling. We evaluate FLICK with several services (an HTTP load-balancer, a Memcached router and a Hadoop data aggregator), showing that it achieves good performance while reducing development effort.

1 Introduction

Distributed applications in data centres increasingly want to adapt networks to their requirements. *Application-specific network services*, such as application load-balancers [23, 40], request data caches [36], and in-network data aggregators [29], therefore blur the boundary between the network fabric at the core and applications at the edge. For example, a Memcached request router can transparently scale deployments by routing requests using knowledge of the Memcached protocol [36]. *In this paper, we explore how application developers, not network engineers, can be supported when implementing new application-specific network services.*

Existing software middlebox platforms, such as ClickOS [30], xOMB [3] and SmartSwitch [53], support only *application-independent* network services, i.e. IP routers, firewalls or transport-layer gateways. Using them to interact with payload data in network flows leads to an impedance mismatch due to their byte-oriented, per-packet APIs. Instead, application developers would prefer high-level constructs and data types when expressing processing logic. For example, when defining the dispatching logic of a Memcached request router, a developer would ideally treat key/value pairs as a first-class data type in their program.

Today's middlebox platforms also force developers to optimise their code carefully to achieve high throughput—implementing a new Click module [24, 30] in C++ that can process data at 10 Gbps line rate is challenging. As a result, many new application-specific network services [40, 29] are built from scratch rather than leveraging the above platforms.

Considerable work went into developing new high-level languages for network control within software-defined networking (SDN) [16, 33, 8, 48]. While these simplify the specification of network management policies, they typically operate on a per-packet basis and support a limited set of per-packet actions once matched, e.g. forwarding, cloning or dropping. In contrast, application-specific network services must refer to payload data, e.g. messages, key/value pairs or deserialised objects, and carry out richer computations, e.g. arbitrary payload transformations, caching or data aggregation.

Our goal is to enable developers to express application-specific network services in a natural high-level programming model, while executing such programs in an efficient and scalable manner. This is challenging for several reasons: (i) in many cases, the cost of data deserialisation and dynamic memory management reduces

Authors are ordered alphabetically, grouped by institution and non-faculty/faculty status.

achievable processing throughput. While high-level programming languages such as Java or Python can manipulate complex application objects, they struggle to provide predictable processing throughput for line-rate processing of network data; (ii) a typical data centre may host hundreds of applications, with each potentially requiring its own network service. Services must thus share resources, e.g. CPU and memory, without interference. Existing middlebox platforms use coarse-grained virtualisation [30], which carries a context-switching overhead of hundreds of microseconds. This is too high for fine-grained resource sharing between many application-specific network services; and (iii) most of the applications use TCP for transport, and an application-specific middlebox needs to terminate TCP connections to access data. Performance and scalability of such middleboxes are often bounded by the high cost of connection termination and frequent socket reads/writes.

We describe **FLICK**, a framework for developers to program and execute application-specific network services. It consists of the FLICK *language* for defining network services, and the FLICK *platform* for executing compiled programs efficiently on multi-core CPUs.

Programs in the FLICK language have *bounded resource usage* and are *guaranteed to terminate*. This is possible because most application-specific network services follow a similar pattern: they deserialise and access application data types, iterate over these data types to perform computation, and output the results as network flows. The language is therefore statically typed, and all built-in types (e.g. `integer`, `string`, and `array`) must have a maximum size to avoid dynamic memory allocation. Programs can refer to complex application-defined data types, such as messages or key/value pairs, for which efficient parsers are synthesised from the type definitions in the program. Since functions can only perform finite iteration over fixed-length data types, FLICK programs with finite input must terminate.

A compiler translates FLICK programs into *task graphs* implemented in C++. Tasks graphs are designed to permit the efficient and safe execution of many concurrent network services on a shared platform. A task graph consists of parallel *tasks* that define the computation of the FLICK program, and *channels* that propagate data between concurrently executing tasks. *Input/output tasks* perform the serialisation/deserialisation of data to and from application objects. Since FLICK programs explicitly specify accesses to application data fields, the compiler can generate custom parsing code, eliminating the overheads of general-purpose parsers.

The FLICK *platform* executes multiple task graphs belonging to different services. To reduce the overhead of frequent connection termination and socket operation, task graphs use a modified version of a highly-scalable user-space TCP stack (mTCP [21]) with Intel's Data Plane Development Kit (DPDK) [20]. Task graphs are also scheduled *cooperatively*, avoiding context-switching overhead. They cannot interfere with each other, both in terms of performance and resources, due to their safe construction from FLICK programs.

We evaluate a prototype implementation of FLICK using both micro-benchmarks and three application-specific network services: an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator. Our results show that FLICK can execute these services with throughput and latency that matches that of specialised middlebox implementations. In addition, it scales with a larger number of compute tasks. This paper focuses on the design, implementation and performance of a single FLICK middlebox. However, the wider vision is of a number of such boxes within a data centre [10].

2 Application-Specific Network Services

FLICK focuses on a specific context: data centres in which multiple, complex, distributed applications run concurrently. In this case, to achieve higher performance, flexibility or efficiency, it is advantageous to execute portions of these applications, e.g. related to load-balancing, caching or aggregation, as *application-specific network services* directly on network elements.

To do this, application developers must add code to network elements such as *software middleboxes*. Today this typically means that they must implement complicated features of the underlying network protocols (e.g. TCP flow construction, HTTP parsing and application data deserialisation). For performance reasons, network services must be highly parallel, which requires considerable developer expertise to achieve. Network resources are also inherently shared: even if hosts can be assigned to single applications, network elements must host many services for different applications.

The goal of FLICK is to allow developers to easily and efficiently introduce application-specific processing into network elements. Present approaches are unsatisfactory for three key reasons: (i) they provide only low-level APIs that focus on the manipulation of individual packets, or at best, individual flows; (ii) they do not permit developers to implement services in high-level languages, but typically rely on the use of low-level languages such as C; and (iii) they provide little support for the high degrees of concurrency that are required to make network service implementations perform well.

Next we elaborate on some of these challenges as encountered in our example applications (§2.1), and then contrast our approach with existing solutions (§2.2).

2.1 Use cases

We consider three sample uses for application-specific services: HTTP load balancing, Memcached request routing, and Hadoop data aggregation.

HTTP load balancer. To cope with a large number of concurrent requests, server farms employ load balancers as front ends. These are implemented by special-purpose hardware or highly-optimised software stacks and both sacrifice flexibility for performance. As a result, load balancers must often be reimplemented for each application to tailor them to specific needs. For example, this may be necessary to ensure consistency when multiple TCP connections are served by the same server; to improve the efficiency of clusters running Java code, a load balancer may avoid dispatching requests to servers that are currently performing garbage collection [27]; finally, there is increasing interest from Internet companies to monitor application-specific request statistics—a task that load balancers are ideally placed to carry out [13].

Memcached proxy. Memcached [15] is a popular distributed in-memory key/value store for reducing the number of client reads from external data sources by caching read results in memory. In production environments, a proxy such as *twemproxy* [52] or *mcrouter* [36] is situated usually between clients and servers to handle key/value mappings and instance configurations. This decouples clients and servers and allows the servers to scale out or in horizontally.

Past attempts to implement Memcached routers have involved user-space solutions [36], incurring high overheads due to expensive memory copies between kernel- and user-space. More recent proposals, such as *MemSwitch* [53], have shown that a dedicated single-purpose software switch that intercepts and processes Memcached traffic can be more efficient. To customise MemSwitch, developers, however, must write complex in-network programs that process raw packet payloads. This not only compromises the safety and performance of the network stack, but also complicates development—it requires knowledge about low-level details of networking as well as skills for writing high-performance, parallelisable packet-processing code.

Hadoop data aggregator. Hadoop [54] is a popular map/reduce framework for data analysis. In many deployments, job completion times are network-bound due to the shuffle phase [9]. This means that performance can be improved through an application-specific network service for *in-network data aggregation* [29], which executes an intermediate in-network reduction within the network topology before data reaches the reducers, thus reducing traffic crossing the network.

Providing an in-network data aggregation for Hadoop serves as a good example of an application-specific service that must carry out complex data serialisation and deserialisation. A developer wishing to implement in-network reduce logic must therefore re-implement the logic necessary to reconstruct Hadoop key/value pairs from TCP flows—a difficult and error-prone task.

2.2 Existing solution space

There are several proposals for addressing the challenges identified in the use cases above. We observe that existing solutions typically fit into one of four classes:

(i) Specialised, hand-crafted implementations. Systems such as *netmap* [43, 44] provide for efficient user-space implementations of packet-processing applications. Unfortunately, they offer only low-level abstractions, forcing developers to process individual packets rather than high-level business logic.

(ii) Packet-oriented middleboxes. Frameworks for implementing software middleboxes, such as *ClickOS* [30] and *SmartSwitch* [53], enable high-performance processing of network data and can be used to build higher-level abstractions. However, they fail to support useful high-level language features such as strong and static typing, or simple support for data-parallel processing.

(iii) Network programmability. More recently, we see increasing deployment of *software-defined networking* techniques, usually *OpenFlow* [31]. More advanced technologies have been proposed such as *P4* [8] and *Protocol Oblivious Forwarding* [47]. These enable efficient in-network processing of traffic, selectively forwarding, rewriting and processing packets. However, they suffer from many of the same issues as (ii) due to their narrow focus on packet-level abstractions.

(iv) Flow-oriented servers. For in-network processing concerned with higher-level flow abstractions, it is common to leverage existing server implementations, such as *Nginx* [35] or *Apache* [51], and customise them either at the source level or through extensibility mechanisms such as modules. Another example is Netflix ribbon [34], which provides a number of highly configurable middlebox services along with a Java library to build custom services. While this raises the level of abstraction somewhat, the overheads of using such large, complex pieces of software to perform application-specific network services are substantial.

3 FLICK Framework

We motivate our design by outlining requirements (§3.1), and providing a high-level overview (§3.2).

3.1 Requirements

Based on the shortcomings of the approaches highlighted in §2.2, we identify the following three design requirements for our framework:

R1: Application-level abstractions: developers should

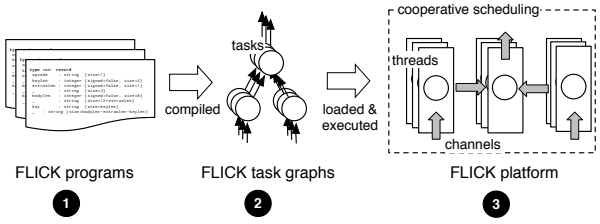


Figure 1: Overview of the FLICK framework

be able to express their network services using familiar constructs and abstractions without worrying about the low-level details of per-packet (or per-flow) processing;

R2: High parallelism: to achieve line-rate performance, programs for application-specific network services must exploit both data and task parallelism without requiring significant effort from the developers;

R3: Safe and efficient resource sharing: middleboxes are shared by multiple applications/users, therefore, we need to ensure that programs do not interfere with one another, both in terms of CPU and memory resources.

To meet these requirements, FLICK follows the scheme shown in Figure 1. For the desired level of abstraction (**R1**), it provides a novel high-level language (❶; §4). The language allows developers to focus on the business logic of their network services ignoring low-level details (e.g. serialisation or TCP reassembly).

Compared to general-purpose languages such as C or Java, the FLICK language offers a constrained programming environment. This makes it easier to compile FLICK programs to parallel FLICK *task graphs* (❷; §5). The division of programs into tasks allows the platform to take advantage of both data and task parallelism, thus exploiting multi-core CPUs (**R2**).

Finally, the FLICK language bounds the resource usage for each invocation of a network service. This allows task graphs to be executed by the FLICK *platform* according to a cooperative scheduling discipline (❸; §5), permitting a large number of concurrent task graphs to share the same hardware resources with little interference (**R3**). A pool of worker threads execute *tasks* cooperatively, while *channels* move data between tasks.

3.2 Overview

We now give a more detailed overview of how a developer uses the FLICK framework (see Figure 1). First they write the logic of their application-specific network services in the FLICK language. After compilation by the FLICK compiler, the FLICK platform runs a program as an *instance*, consisting of a set of *task graphs*. Each task graph comprises of a directed acyclic graph of *tasks* connected by *task channels*. Depending on the program semantics, multiple instances of the task graph can be instantiated for each network request, or a single graph can be used by multiple requests.

A task is a schedulable unit of computation. Each task processes a stream of input values and generates a stream of output values. Initial input to the task graph is handled by one or more *input tasks*, which consume data from a single *input channel*, i.e. the byte stream of a TCP connection. An input task then deserialises bytes to values using deserialisation/parsing code generated by the FLICK compiler from the types specified in the FLICK program. Deserialisation splits data into the smallest units appropriate for the task being considered. For example, if the input is from a web client, the byte stream would be deserialised into individual complete HTTP requests; for Hadoop, a key/value pair is more appropriate.

Received data is then processed by one or more compute tasks and, finally, output from the task graph is emitted to the outside world via an *output task*, representing a single outgoing TCP connection. The output task also executes efficient serialisation code generated from the FLICK program, converting values into a byte stream that is placed onto an *output channel* for transmission.

4 FLICK Programming Model

FLICK provides a domain-specific language (DSL) targeting application-specific middlebox programming. While a difficult task, we decided to design a new language because we found existing general-purpose languages inappropriate for middlebox programming due to their excessive expressive power. Even safe redesigns of widely-used languages, such as *Cyclone* [22], are too powerful for our needs because, by design, they do not restrict the semantics of programs to terminate and bound the used resources. Existing specialised languages for network services, such as *PLAN* [18], are typically packet-centric. This makes it hard to implement application-specific traffic logic that is flow-centric. A new domain-specific language presents us with the opportunity to incorporate primitive abstractions that better fit the middlebox domain.

We also considered *restricting* an existing language to suit our needs (for example OCaml restricted so it performs no unbounded loops and no garbage collection). This, however, presented two difficulties: (i) exposing programmers to a familiar language but with altered semantics would be confusing; and (ii) it would prevent us from including language features for improved safety, such as static type-checking.

Numerous systems programming tasks have been simplified by providing DSLs to replace general-purpose programming languages [26, 28, 6, 39, 11]. The FLICK language is designed (i) to provide convenient, familiar high-level language abstractions targeted specifically at middlebox development, e.g. application-level types, processes and channels alongside traditional functions and primitive types; (ii) to take advantage of execution

Listing 1: FLICK program for Memcached proxy

```
1  type cmd: record
2    key   : string
3
4  proc Memcached: (cmd/cmd client, [cmd/cmd] backends)
5    | backends => client
6    | client => target_backend(backends)
7
8  fun target_backend: ([-/cmd] backends, req:cmd) -> ()
9    let target = hash(req.key) mod len(backends)
10   req => backends[target]
```

parallelism for high throughput; and (iii) to enable efficient and safe handling of multiple programs and many requests on shared hardware resources by making it impossible to express programs with undesirable behaviour, such as unbounded resource consumption.

In the FLICK language, developers describe application-specific network services as a collection of interconnected *processes*. Each process manipulates values of the application's data types, in contrast to earlier work which described network services as simple packet processors [24, 7, 30]. Application data is carried over *channels*, which interconnect processes with one another and with network flows. Processes interact with channels by consuming and processing input data read from them, and by transmitting output over them. Processes, channels and network interactions are handled by the FLICK *platform*.

The FLICK language is designed to achieve efficient parallel execution on multi-core CPUs using high-level parallel primitives. By default, the language offers parallelism across multiple requests, handling them concurrently. It supports the safe sharing of resources by *bounding* the resource use of an individual program. Processing of continuous network flows belonging to an application is subdivided into discrete units of work so that each process consumes only a bounded amount of resource. To achieve this, FLICK control structures are restricted to finite iteration only. This is not a significant limitation, however, as application-specific network services typically carry out deterministic transformations of network requests to generate responses. User-defined functions are written in FLICK itself, rather than in a general purpose language as in Click [24] or Pig [37]), which preserves the safety of network services expressed in FLICK.

After presenting the FLICK language by example (§4.1), we describe its application data types (§4.2), primitives and compilation (§4.3).

4.1 Overview

Listing 1 shows a sample FLICK program that implements a Memcached proxy. Programs are composed of three types of declarations: *data types* (lines 1–2), *processes* (lines 4–6) and *functions* (lines 8–10).

Processes have signatures that specify how they connect to the outside world. In this case, a process called `Memcached` declares a signature containing two channels (line 4): the `client` channel produces and accepts values of type `cmd`, while `backends` is an array of channels, each of which produces and accepts values of type `cmd`.

Processes are instantiated by the FLICK platform, which binds channels to underlying network flows (§5). In this example, when a client sends a request, the FLICK platform creates a new `Memcached` task graph and assigns the client connection to this graph. Giving each client connection a new task graph ensures that responses are routed back to the correct client.

A process body describes how data is transformed and routed between channels connected to a process. The language design ensures that only a finite amount of input from each channel is consumed. The body of the `Memcached` process describes the application-specific network service: data received from *any* channel in `backends` is sent to the `client` channel (line 5); data received from the client is processed by the `target_backend` function (line 6), which in turn writes to a suitable channel in the `backends` array (line 10).

4.2 Supporting application data types

FLICK programs operate on application data types representing the exchanged messages. After an input task reads such messages from the network, they are parsed into FLICK data types. Similarly, before processed data values are transmitted by an output task, they are serialised into the appropriate wire format representation.

The transformation of messages between wire format and FLICK data types is defined as a *message grammar*. During compilation, FLICK generates the corresponding parsing and serialisation code from the grammar, which is then used in the input and output tasks of the task graph, respectively. The generated code is optimised for efficiency in three ways: (i) it does not dynamically allocate memory; (ii) it supports the incremental parsing of messages as new data arrives; and (iii) it is adapted automatically to specific use cases.

The syntax to define message grammars is based on that of the *Spicy* (formerly *Binpac++* [46]) parser generator. The language provides constructs to define messages and their serialised representation through units, fields, and variables, and their composition: *units* are used to modularise grammars; *fields* describe the structure of a unit; and *variables* can compute the value of expressions during parsing or serialisation, e.g. to determine the size of a field. FLICK grammars can express any LL(1)-parsable grammar as well as grammars with dependent fields, in a manner similar to Spicy. The FLICK framework provides reusable grammars for common protocols, such as the HTTP [14] and Memcached protocols [50]. Developers can also specify additional mes-

Listing 2: Partial grammar for Memcached protocol

```
1  type cmd = unit {
2      %byteorder = big;
3
4      magic_code          : uint8;
5      opcode              : uint8;
6      key_len             : uint16;
7      extras_len          : uint8;
8                          : uint8;  # anonymous field,
                                reserved for future use
9      status_or_v_bucket  : uint16;
10     total_len           : uint32;
11     opaque              : uint32;
12     cas                 : uint64;
13
14     var value_len       : uint32
15         &parse = self.total_len -
16             (self.extras_len + self.key_len)
17         &serialize = self.total_len =
18             self.key_len + self.extras_len + $$;
19     extras   : bytes &length = self.extras_len;
20     key      : string &length = self.key_len;
21     value    : bytes &length = self.value_len;
22 };
```

Listing 3: FLICK program for Hadoop data aggregator

```
1  type kv: record
2    key : string
3    value : string
4
5  proc hadoop: ([kv/-] mappers, -/kv reducer):
6    if (all_ready(mappers)):
7     let result = foldt on mappers
8       ordering elem e1, e2 by elem.key as e_key:
9      let v = combine(e1.val, e2.val)
10     kv(e_key, v)
11    result => reducer
12
13 fun combine(v1: string, v2: string) -> (string): ...
```

sage grammars for custom formats, such as application-specific Hadoop data types.

Listing 2 shows a simplified grammar for Memcached. The `cmd` unit for the corresponding FLICK data type is a sequence of fixed-size fields (lines 4–12), a variable (lines 14–18), and variable-size fields (lines 19–21). Each field is declared with its wire-format data type, e.g. the `opcode` field is an 8-bit integer (line 5). The sizes of the `extras`, `key`, and `value` fields are determined by the parsed value of the `extras_len` and `key_len` fields as well as the `value_len` variable, which is computed during parsing according to the expression in lines 15 and 16. During serialisation, the values of `extras_len`, `key_len`, and `value_len` are updated according to the sizes of the values stored in the `extras`, `key`, and `value` fields. Subsequently, the value of `total_len` is updated according to the variable's serialisation expression in lines 17 and 18. The `%byteorder` property declaration in line 2 specifies the wire format encoding of number values—the generated code transforms such values between the specified big-endian encoding and the host byte-order. More advanced features of the grammar language include choices between alternative field sequences, field repetitions (i.e. lists), and transformations into custom FLICK field types (e.g. enumerations).

FLICK grammars aim to be reusable and thus include all fields of a given message format, even though application-specific network services often only require a subset of the information encoded in a message. To avoid generated parsers and serialisers handling unnecessary data, FLICK programs make accesses to message fields explicit by declaring a FLICK data type corresponding to the message (Listing 1, lines 1–2). This enables the FLICK compiler to generate input and output tasks that only parse and serialise the required fields for these data types and their dependencies. Other fields are aggregated into either simplified or composite fields, and then skipped or simply copied in their wire format representation. Developers can thus reuse complete message grammars to generate parsers and serialisers, while benefiting from efficient execution for their application-specific network service.

The current FLICK implementation does not support exceptions, but data type grammars could provide a default behaviour when a message is incomplete or not in an expected form.

4.3 Primitives and compilation

The FLICK language is strongly-typed for safety. To facilitate middlebox programming, it includes channels, processes, explicit parallelism, and exception handling as native features. For example, events such as broken connections can be caught and handled by FLICK functions, which can notify a backend or record to a log. State handling is essential for describing many middleboxes, and the language supports both session-level and long-term state, whose scope extends across sessions. The latter is provided through a key/value abstraction to task graph instances by the FLICK platform. To access it, the programmer declares a dictionary and labels it with a `global` qualifier. Multiple instances of the service share the key/value store.

The language is restricted to allow only computations that are guaranteed to terminate, thus avoiding expensive isolation mechanisms while supporting multiple processes competing for shared resources. This restriction allows static allocation of memory and cooperative task scheduling (see §5).

The FLICK language offers primitives to support common datatypes such as bytes, lists and records. Iteration may only be carried out on finite structures (e.g. lists). It also provides primitives such as `fold`, `map` and `filter` but it does not offer higher-order functions: functions such as `fold` are translated into finite for-loops. Datatypes may be annotated with cardinalities to determine statically the required memory. Loops and branching are compiled to their native counterparts in C++. Channel- and process-related code is translated to API calls exposed by the platform (see §5). The language re-

lies on the C++ compiler to optimise the target code.

Channels are typed, and at compile time the platform determines that FLICK programs only send valid data into channels. Due to the language's static memory restrictions, additional channels cannot be declared at runtime, though channels may be rebound, e.g. to connect to a different backend server.

The language also provides `foldt`, a parallel version of fold that operates over a *set* of channels. This allows the efficient expression of typical data processing operations, such as a *k*-way merge sort in which sorted streams of keys from *k* channels are combined by selecting elements with the smallest key. The expression `foldt f o cs` aggregates elements from an array of channels `cs`, selecting elements according to a function `o` and aggregating according to a function `f`. As `f` must be commutative and associative, the aggregation can be performed in parallel, combining elements in a pair-wise manner until only the result remains.

As shown in Listing 3, the `foldt` primitive can be used to implement an application-level network service for parallel data aggregation in Hadoop. Whenever key/value pairs become available from the mappers (lines 5–6), `foldt` is invoked (lines 7–10). Elements `elem` are ordered based on `elem.key` (line 8), and values of elements with the same key (`e_key`) are merged using a combine function (line 9) to create a new key/value pair (line 10).While `foldt` could be expressed using core language primitives, the FLICK platform has a custom implementation for performance reasons.

While designed to achieve higher safety and performance, the constraints introduced in the design of the FLICK language, e.g. the lack of support for unbounded computation or dynamic memory allocation, imply that not all possible computations can be expressed in FLICK. For instance, algorithms requiring loops with unbounded iterations (e.g. `while`-like loops) cannot be encoded. In a general purpose programming language, this would be a severe constraint but for the middlebox functionality that FLICK targets we have not found this to cause major limitations.

5 FLICK Platform

The FLICK platform is designed around a *task graph* abstraction, composed of tasks that deserialise input data to typed values, compute over those values, and serialise results for onward transmission. The FLICK compiler translates an input FLICK program to C++, which is in turn compiled and linked against the platform runtime for execution. Figure 2 shows an overview of the FLICK platform, which handles network connections, the task graph life-cycle, the communication between tasks and the assignment of tasks to worker threads. Task graphs exploit task and data parallelism at runtime as tasks are

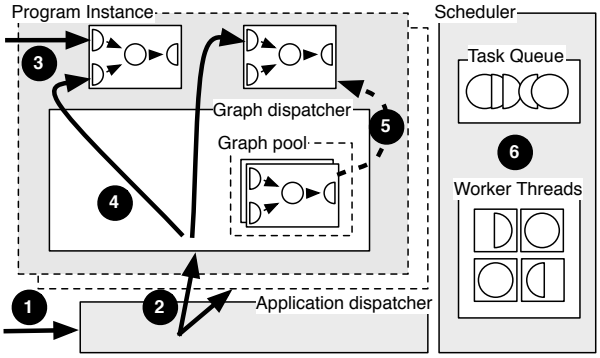


Figure 2: Main components of the FLICK platform

assigned to worker threads. Even with only one large network flow, serialisation, processing and deserialisation tasks can be scheduled to run on different CPU cores.

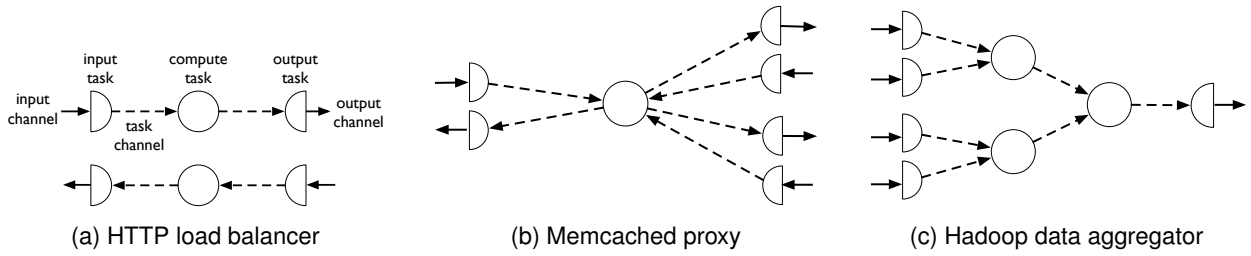
(i) The *application dispatcher* manages the life-cycle of TCP connections: first it maps new incoming connections ❶ to a specific program instance ❷, typically based on the destination port number of the incoming connection. The application dispatcher manages the listening sockets that handle incoming connections, creating a new input channel for each incoming connection and handing off data from that connection to the correct instance. When a client closes an input TCP connection, the application dispatcher indicates this to the instance; when a task graph has no more active input channels, it is shut down. New connections are directly connected to existing task graphs ❸.

(ii) The *graph dispatcher* assigns incoming connections to task graphs ❹, instantiating a new one if none suitable exists. The platform maintains a pre-allocated pool of task graphs to avoid the overhead of construction ❺. The graph dispatcher also creates new output channel connections to forward processed traffic.

(iii) Tasks are cooperatively scheduled by the *scheduler*, which allocates work among a fixed number of worker threads ❻. The number of worker threads is determined by the number of CPU cores available, and worker threads are pinned to CPU cores.

Tasks in a task graph become runnable after receiving data in their input queues (either from the network or from another task). A task that is not currently executing or scheduled is added to a worker queue when it becomes runnable. All buffers are drawn from a pre-allocated pool to avoid dynamic memory allocation. Input tasks use non-blocking sockets and `epoll` event handlers to process socket events. When a socket becomes readable, the input task attached to the relevant socket is scheduled to handle the event.

For scheduling, each worker thread is associated with its own FIFO task queue. Each task within a task graph has a unique identifier, and a hash over this identifier de-



(a) HTTP load balancer (b) Memcached proxy (c) Hadoop data aggregator

Figure 3: Task graphs for different application-specific network services

termines which worker's task queue the task should be assigned to. When a task is to be scheduled, it is always added to the same queue to reduce cache misses.

Each worker thread picks a task from its own queue. If its queue is empty, the worker attempts to scavenge work from other queues and, if none is found, it sleeps until new work arrives. A worker thread runs a task until either all its input data is consumed, or it exceeds a system-defined time quantum, the *timeslice threshold* (typically, 10–100 μs; see §6). If the timeslice threshold is exceeded, the code generated by the FLICK compiler guarantees that the task re-enters the scheduler, placing itself at the back of the queue if it has remaining work to do. A task with no work is not added to the task queue, but when new items arrive in its input channels, it is scheduled again.

A disadvantage of allocating tasks belonging to the same task graphs onto different CPU cores is that this would incur several cache invalidations as data move from one core to another. On the other hand, our design enables higher parallelism as different tasks can execute concurrently in a pipelined fashion, leading to higher throughput.

Some middlebox services must handle many concurrent connections, and they frequently write and read small amounts of data. The kernel TCP stack has a high overhead for creating and destroying sockets to support the Linux Virtual File System (VFS) interface [17]. Socket APIs also require switching between user- and kernel-mode, which adds further overhead. As a result, the FLICK platform uses mTCP [21], a highly scalable user-space TCP stack, combined with Intel's DPDK [20] to reduce these overheads. The original mTCP implementation did not support multi-threaded applications, and we modified mTCP so that Flick I/O tasks can access sockets independently. To take utilise the efficient DPDK runtime environment, mTCP executes as a DPDK task. All of these optimisations, significantly improve performance for network-bound services (see §6.3).

6 Evaluation

The goals of our evaluation are to investigate whether the high-level programming abstraction that FLICK carries a performance and scalability cost and whether DPDK and mTCP improve performance. We implement FLICK programs for the use cases introduced in §2.1, i.e. an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator, and compare their performance against baselines from existing implementations.

After describing the implementation of our use cases (§6.1) and the experimental set-up (§6.2), we explore the performance and scalability of FLICK (§6.3). After that, we examine how well the FLICK platform isolates resource consumption of multiple FLICK programs using cooperative scheduling (§6.4).

6.1 Use case implementation

For our three use cases, Figure 3 shows the task graph obtained from the corresponding FLICK program.

HTTP load balancer. This FLICK program implements an HTTP load balancer that forwards each incoming HTTP request to one of a number of backend web servers. Forwarding is based on a naive hash of the source IP and port and destination IP and port. Figure 3a shows the corresponding task graph. The application dispatcher forwards each new TCP connection received on port 80 to the graph dispatcher. The graph dispatcher creates a new task graph, which is later destroyed when the connection closes. The input task deserialises the incoming data into HTTP requests. For the first request, the compute task calculates a hash value selecting a backend server for the request. Subsequent requests on the same connection are forwarded to the same backend server. On their return path no computation or parsing is needed, and the data is forwarded without change. We also implement a variant of the HTTP load balancer that does not use backend servers but which returns a fixed response to a given request. This is effectively a static web server, which we use to test the system without backends.

Memcached proxy. In this use case, the FLICK program (Listing 1) receives Memcached look-up requests for keys. Requests are forwarded based on hash partitioning to a set of Memcached servers, each storing a disjoint section of the key space. Responses received from the Memcached servers are returned to clients.

Figure 3b shows the corresponding task graph. As before, a new task graph is created for each new TCP connection. Unlike the HTTP load balancer, requests

from the same client can be dispatched to different Memcached servers, which means that the compute task must have a fan-out greater than one.

When a request is received on the input channel, it is deserialised by the input task. The deserialisation code is automatically generated from the `type` specification in Listing 2. The deserialiser task outputs the Memcached request object, containing the request keys and body, which are passed on to the compute task. The compute task implements the dispatching logic. It identifies the Memcached server responsible for that key and forwards the request to it through the serialiser task. When the response is received from the Memcached server, the deserialiser task deserialises it and passes the response object to the compute task, which returns it to the client through the serialiser task.

Hadoop data aggregator. The Hadoop data aggregator implements the combiner function of a map/reduce job to perform early data aggregation in the network, as described in §2.1. It is implemented in FLICK according to Listing 3. We focus on a wordcount job in which the combiner function aggregates word counters produced by mappers over a set of documents.

For each Hadoop job, the platform creates a separate task graph per reducer (Figure 3c). The input tasks deserialise the stream of intermediate results (i.e. key/value pairs) from the mappers. Compute tasks combine the data with each compute task taking two input streams and producing one output. The output task converts the data to the byte stream, as per the Hadoop wire format.

6.2 Experimental set-up

We deploy the prototype implementation of the FLICK platform on servers with two 8-core Xeon E5-2690 CPUs running at 2.9 Ghz with 32 GB of memory. Clients and back-end machines are deployed on a cluster of 16 machines with 4-core Xeon E3-1240 CPUs running at 3.3 Ghz. All machines use Ubuntu Linux version 12.04. The clients and backend machines have 1 Gbps NICs, and the servers executing the FLICK platform have 10 Gbps NICs. The client and backend machines connect to a 1 Gbps switch, and the FLICK platform connects to a 10 Gbps switch. The switches have a 20 Gbps connection between them. We examine the performance of FLICK with and without mTCP/DPDK.

To evaluate the performance of the HTTP load balancer, we use multiple instances of *ApacheBench* (ab) [4], a standard tool for measuring web server performance, together with 10 backend servers that run the *Apache* web server [51]. Throughput is measured in terms of connections per second as well as requests per second for HTTP keep-alive connections. We compare against the standard Apache (*mod_proxy_balancer*) and the Nginx [35] load balancers.

For the Memcached proxy, we deploy 128 clients running *libmemcached* [1], a standard client library for interacting with Memcached servers. We use 10 Memcached servers as backends and compare the performance against a production Memcached proxy, *Moxi* [32]. We measure performance in terms of throughput (i.e. requests served per second) and request latency. Clients send a single request and wait for a response before sending the next request.

For the Hadoop data aggregator, the workload is a *wordcount* job. It uses a sum as the aggregation computation and an input dataset with a high data reduction ratio. The datasets used in experiments are 8 GB, 12 GB and 16 GB (larger data sets were also used for validation). Here we measure performance in terms of the absolute network throughput.

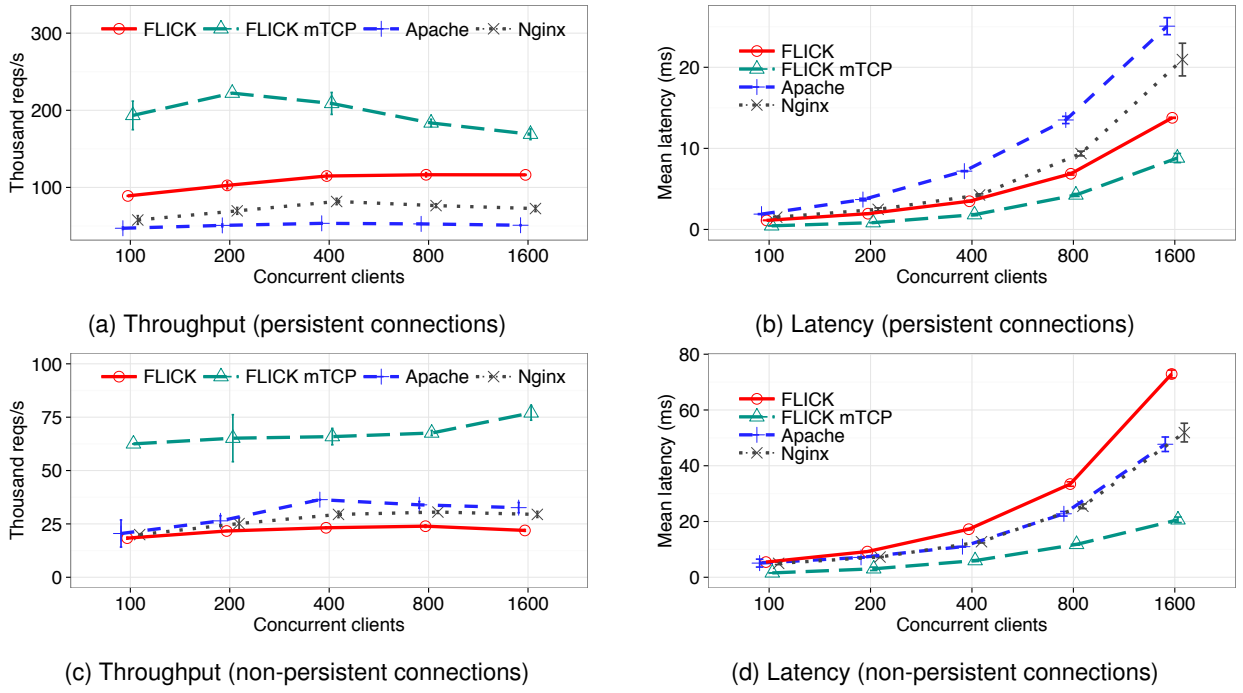
In all graphs, the plotted points are the mean of five runs with identical parameters. Error bars correspond to a 95% confidence interval.

6.3 Performance

HTTP load balancer. We begin by measuring the performance of the static web server with an increasing load. This exercises the following components of the FLICK platform: HTTP parsing, internal and external channel operation and task scheduling. The results are for 100 to 1,600 concurrent connections (above these loads, Apache and Nginx begin to suffer timeouts). Across the entire workload, FLICK achieves superior performance. It achieves a peak throughput of 306,000 requests/sec for the kernel version and 380,000 requests/sec with mTCP. The maximum throughput achieved by Apache is 159,000 requests/sec and by Nginx is 217,000 requests/sec. FLICK also shows lower latency, particularly at high concurrency when Apache and Nginx use large numbers of threads. This confirms that, while FLICK provides a general-purpose platform for creating application-specific network functions, it can outperform purpose-written services.

To investigate the per-flow overhead due to TCP set-up/tear-down, we also repeat the same experiment but with each web request establishing a separate TCP connection (i.e. non-persistent HTTP). This reduces the throughput in all deployments: 35,000 requests/sec for Apache; 44,000 requests/sec for Nginx; and 45,000 requests/sec for FLICK, which maintains the lowest latency. Here the kernel TCP performance for connection set-up and tear-down is a bottleneck: the mTCP version of FLICK handles up to 193,000 requests/sec.

Next, we repeat the experiment using our HTTP load balancer implementation to explore the impact of both receiving and forwarding requests. The set-up is as described in §6.2. We use small HTTP payloads (137 bytes each) to ensure that the network and the backends are



(a) Throughput (persistent connections)

(b) Latency (persistent connections)

(c) Throughput (non-persistent connections)

(d) Latency (non-persistent connections)

Figure 4: HTTP load balancer throughput and latency for an increasing number of concurrent connections

never the bottleneck. As for the web server experiment, we first consider persistent connections. Figures 4a and 4b confirm the previous results: FLICK achieves up to 1.4× higher throughput than Nginx and 2.2× higher than Apache. Using mTCP, the performance is even better with higher throughput and lower latency: FLICK achieves a maximum throughput 2.7× higher than Nginx and 4.2× higher than Apache. In all cases, FLICK has lower latency.

With non-persistent connections, the kernel version of FLICK exhibits a lower throughput than Apache and Nginx (see Figure 4c). Both Apache and Nginx keep persistent TCP connections to the backends, but FLICK does not, which increases its connection set-up/tear-down cost. When mTCP is used with its lower per connection cost, FLICK shows better performance than either with a maximum throughput 2.5× higher than that of Nginx and 2.1× higher than that of Apache. In addition, both the kernel and mTCP versions of FLICK maintain the lowest latency of the systems, as shown in Figure 4d.

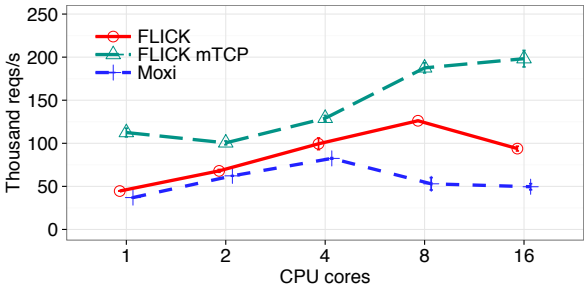
Memcached proxy. For the Memcached proxy use case, we compare the performance of FLICK against *Moxi* [32], as we increase the number of CPU cores. We chose *Moxi* because it supports the binary Memcached protocol and is multi-threaded. In our set-up, 128 clients make concurrent requests using the Memcached binary protocol over persistent connections, which are then multiplexed to the backends.

Figures 5a and 5b show the throughput, in terms of the number of requests/sec, and the latency, respectively. With more CPU cores, the throughout initially increases for both systems. The kernel version achieves a maximum throughput of 126,000 requests/sec with 8 CPU cores and the mTCP version achieves 198,000 requests/sec with 16 CPU cores. Moxi peaks at 82,000 requests/sec with 4 CPU cores. FLICK's latency decreases with more CPU cores due to the larger processing capacity available in the system. The latency of Moxi beyond 4 CPU cores and FLICK's beyond 8 CPU cores increases as threads compete over common data structures.

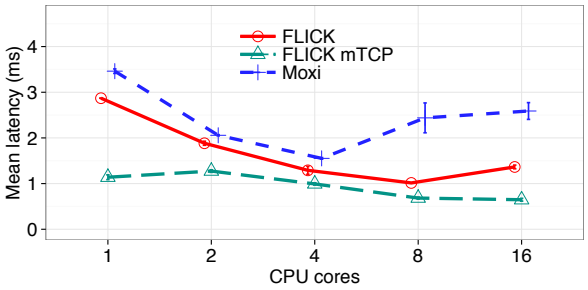
Hadoop data aggregator. The previous use cases had relatively simple task graphs (see Figure 3) and considerable overhead comes from the connection set-up and tear-down, with many network requests processed in parallel. In contrast, the Hadoop data aggregator use case has a more complex task graph, and we use it to assess the overhead of FLICK's communication channels and intra-graph scheduling. Here the tasks are compute bound, and the impact of the network overhead is limited. We only present the kernel results because the mTCP results are similar.

We deploy 8 mappers clients, each with 1 Gbps connections, to connect to the FLICK server. The task graph therefore has 16 tasks (8 input, 7 processing and 1 output). The FLICK Hadoop data aggregator runs on a server with 16 CPU cores without hyper-threading.

Figure 6 shows that FLICK scales well with the number of CPU cores, achieving a maximum throughput of 7,513 Mbps with 16 CPU cores. This is the maximum capacity of the 8 network links (once accounted for TCP



(a) Throughput



(b) Latency

Figure 5: Memcached proxy throughput and latency versus number of CPU cores used

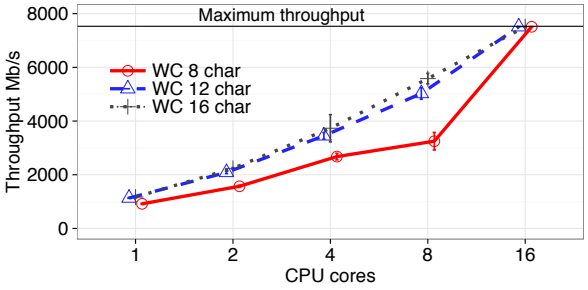


Figure 6: Median throughput for Hadoop data aggregator versus number of CPU cores

overhead), and matches measurements from `iperf`. We conclude that the FLICK platform can exploit the high level of parallelism of multi-core servers and efficiently schedule multiple tasks concurrently to maximise network throughput.

The results in Figure 6 represent three data sets of 8 GB, 12 GB and 16 GB mentioned in §6.2, consisting of words of 8, 12 and 16 characters, respectively. The FLICK platform can more efficiently process the longer words because they comprise fewer key value pairs.

6.4 Resource sharing

We finish our experiments by examining the ability of the FLICK platform to ensure efficient resource sharing, as described in §3.1. For this, we use a micro-benchmark running 200 tasks. Each task consumes a finite number of data items, computing a simple addition for each input byte. The tasks are equally split between two classes:

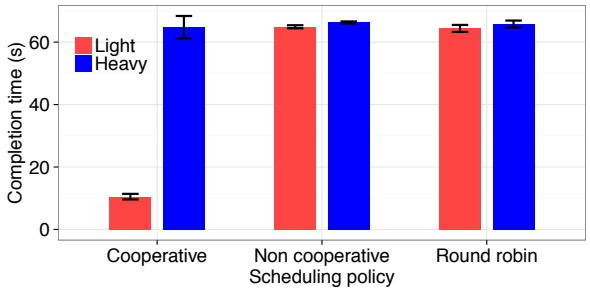


Figure 7: Completion time for "light" and "heavy" tasks with three scheduling policies

light tasks operate on 1 KB data items; and *heavy* tasks operate on 16 KB data items. We consider three scheduling policies: (i) *cooperative* is the policy used by FLICK, in which each task is given a fixed amount of CPU time before it yields control to another task; (ii) *non-cooperative* runs a scheduled task to completion, potentially letting the OS scheduler preempt it; and (iii) *round robin* schedules each task for one data item only.

Figure 7 shows the total completion time for light and heavy tasks. Since the light tasks handle less data, they should, given a fair share of resources, finish before the heavy tasks. With the *round robin* policy, this does not happen: the heavy tasks take longer to process one data item. Each time they are scheduled, they occupy the worker thread for longer than a light task. Conversely, with the *non-cooperative* policy, each task runs to completion. The total completion time for the light and heavy tasks is determined by their scheduling order. However, with FLICK's *cooperative* policy, the light tasks are allowed to complete ahead of the heavy tasks without increasing the overall runtime—each task is given a fair share of the CPU time.

7 Related Work

Network programming languages are essential to the usability and scalability of *software-defined networking* (SDN), allowing high-level configuration logic to be translated to low-level network operations. Inspired by *Frenetic* [16], *NetKAT* [2] is a high-level network programming language based on Kleene algebra [25], in which network policies are compiled into a low-level programming abstraction such as *OpenFlow* [31] flow tables. Similarly, the *Policy Graph Abstraction* (PGA) [41] expresses network policies as a coherent, conflict-free policy set and supports automated, correct and independent composition of middlebox policies. These systems focus on network management and configuration and not on the more expressive programs for application-specific network services that are targeted by FLICK.

FLICK ensures that programs execute in a timely manner by restricting the expressiveness of the programming language. Another possible approach is to explicitly ver-

ify that a specific program meets requirements. This verification apparoach has been used to check simple, stateless Click pipelines [12] but might be harder for more complex middlebox programs.

There are proposed extensions to the packet processing done by OpenFlow. *P4* [8] is a platform- and protocol-independent language for packet processors, which allows the definition of new header fields and protocols for use in match/action tables. *Protocol Oblivious Forwarding* (POF) [47] also provides a flexible means to match against and rewrite packet header fields. *Packet Language for Active Networks* (PLAN) [18] is a stateless and strongly-typed functional language for active networking in which packets carry programs to network nodes for execution. In general, these approaches are limited to expressing control-plane processing of packets in contrast to FLICK, which deals with application layer data.

Software middlebox platforms. Recently network services have been deployed on commodity hardware to reduce costs and increase flexibility. *Click* [24] processes packets through a chain of installed elements, and it supports a wide variety of predefined elements. Programmers, however, must write new elements in C++, which can be error-prone. *ClickOS* [30] combines Click with MiniOS and focuses on the consolidation of multiple software middlebox VMs onto a single server. It overcomes current hypervisor limitations through a redesigned I/O system and by replacing Open vSwitch [38] with a new software switch based on VALE [45]. ClickOS targets packet level processing, e.g. manipulating header fields or filtering packets; FLICK, by contrast, operates at the application level, and the approaches can be seen as orthogonal. It would be challenging for ClickOS to parse and process HTTP data when a single data item may span multiple packets or Memcached data when a packet may contain multiple data items.

Merlin [48] is a language that safely translates policies, expressed as regular expressions for encoding paths, into Click scripts. Similarly, *IN-NET* [49] is an architecture for the deployment of custom in-network processing on ClickOS with an emphasis on static checking for policy safety. In a similar vein, *xOMB* [3] provides a modular processing pipeline with user-defined logic for flow-oriented packet processing. *FlowOS* [7] is a flow-oriented programmable platform for middleboxes using a C API similar to the traditional socket interface. It uses kernel threads to execute flow-processing modules without terminating TCP connections. Similar to ClickOS, these platforms focus on packet processing rather than the application level. *SmartSwitch* [53] is a platform for high-performance middlebox applications built on top of NetVM [19], but it only supports UDP applications, and it does not offer a high-level programming model.

Eden is a platform to execute application-aware network services at the end hosts [5]. It uses a domain-specific language, similar to F#, and enables users to implement different services ranging from load balancing to flow prioritisation. By operating at the end hosts, it limits the set of network services that can be supported. For example, it would be impossible to implement in-network aggregation or in-network caching.

Split/Merge [42] is a hypervisor-level mechanism that allows balanced, stateful elasticity and migration of flow state for virtual middleboxes. Per-flow migration is accomplished by identifying the external state of network flows, which has to be split among replicas. Similar elasticity support could be integrated with FLICK.

8 Conclusions

Existing platforms for in-network processing typically provide a low-level, packet-based API. This makes it hard to implement application-specific network services. In addition, they lack support for low-overhead performance isolation, thus preventing efficient consolidation.

To address these challenges, we have developed FLICK, a domain-specific language and supporting platform that provides developers with high-level primitives to write generic application-specific network services. We described FLICK's programming model and runtime platform. FLICK realises processing logic as restricted cooperatively schedulable tasks, allowing it to exploit the available parallelism of multi-core CPUs. We evaluated FLICK through three representative use cases, an HTTP load balancer, a Memcached proxy and a Hadoop data aggregator. Our results showed that FLICK greatly reduces the development effort, while achieving better performance than specialised middlebox implementations.

References

[1] AKER, B. libmemcached. `http://libmemcached.org/libMemcached.html`.

[2] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic Foundations for Networks. In *Proc. 41th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2014).

[3] ANDERSON, J. W., BRAUD, R., KAPOOR, R., PORTER, G., AND VAHDAT, A. xOMB: Extensible Open Middleboxes with Commodity Servers . In *Proc. 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)* (2012).

[4] APACHE FOUNDATION. Apache HTTP Server Benchmarking Tool, 2015. `https://httpd.apache.org/docs/2.2/programs/ab.html`.

[5] BALLANI, H., COSTA, P., GKANTSIDIS, C., GROSVENOR, M. P., KARAGIANNIS, T., KOROMILAS, L., AND O'SHEA, G. Enabling End-host Network Functions. In *Proc. of ACM SIGCOMM* (2015).

[6] BANGERT, J., AND ZELDOVICH, N. Nail: A Practical Tool for Parsing and Generating Data Formats. In *Proc. 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2014).

[7] BEZAHAF, M., ALIM, A., AND MATHY, L. FlowOS: A Flow-based Platform for Middleboxes. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2013).

[8] BOSSHART, P., DALY, D., GIBB, G., IZZARD, M., MCKEOWN, N., REXFORD, J., SCHLESINGER, C., TALAYCO, D., VAHDAT, A., VARGHESE, G., AND WALKER, D. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review 44*, 3 (2014), 87–95.

[9] CHOWDHURY, M., KANDULA, S., AND STOICA, I. Leveraging Endpoint Flexibility in Data-intensive Clusters. In *Proc. of ACM SIGCOMM* (2013).

[10] COSTA, P., MIGLIAVACCA, M., PIETZUCH, P., AND WOLF, A. L. NaaS: Network-as-a-Service in the Cloud. In *Proc. 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)* (2012).

[11] DAGAND, P.-E., BAUMANN, A., AND ROSCOE, T. Filet-o-fish: Practical and Dependable Domain-specific Languages for OS Development. *SIGOPS Oper. Syst. Rev. 43*, 4 (2010), 35–39.

[12] DOBRESCU, M., AND ARGYRAKI, K. Software dataplane verification. In *Proc. USENIX Networked Systems Design and Implementation (NSDI)* (2014), pp. 101–114.

[13] DONOVAN, S., AND FEAMSTER, N. Intentional Network Monitoring: Finding the Needle without Capturing the Haystack. In *Proc. 13th ACM Workshop on Hot Topics in Networks (HotNets)* (2014).

[14] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1, June 1999. `http://ietf.org/rfc/rfc2616.txt`.

[15] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal 2004*, 124 (2004).

[16] FOSTER, N., HARRISON, R., FREEDMAN, M. J., MONSANTO, C., REXFORD, J., STORY, A., AND WALKER, D. Frenetic: A Network Programming Language. In *Proc. 16th ACM SIGPLAN International Conference on Functional Programming (ICFP)* (2011).

[17] HAN, S., MARSHALL, S., CHUN, B.-G., AND RATNASAMY, S. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proc. 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012).

[18] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., AND NETTLES, S. PLAN: A Packet Language for Active Networks. In *Proc. 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP)* (1998).

[19] HWANG, J., RAMAKRISHNAN, K. K., AND WOOD, T. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[20] INTEL. Intel Data Plane Development Kit (DPDK). http://www.intel.com/go/dpdk, 2014.

[21] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[22] JIM, T., MORRISETT, J. G., GROSSMAN, D., HICKS, M. W., CHENEY, J., AND WANG, Y. Cyclone: A Safe Dialect of C. In *Proc. USENIX Annual Technical Conference (ATC)* (2002).

[23] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. In *Proc. European Conference on Computer Systems (EuroSys)* (2013).

[24] KOHLER, E., MORRIS, R., , CHEN, B., JANNOTTI, J., AND KAASHOEKBENJIE, M. F. The Click Modular Router. *ACM Transactions on Computer Systems 18*, 3 (2000), 263–297.

[25] KOZEN, D., AND SMITH, F. Kleene Algebra with Tests: Completeness and Decidability. In *Proc. 10th International Workshop on Computer Science Logic (CSL)* (1996).

[26] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative Networking. *Commun. ACM 52*, 11 (2009), 87–95.

[27] MAAS, M., ASANOVIĆ, K., HARRIS, T., AND KUBIATOWICZ, J. The Case for the Holistic Language Runtime System. In *Proc. 1st International Workshop on Rack-scale Computing (WRSC)* (2014).

[28] MADHAVAPEDDY, A., HO, A., DEEGAN, T., SCOTT, D., AND SOHAN, R. Melange: Creating a "Functional" Internet. *SIGOPS Oper. Syst. Rev. 41*, 3 (2007), 101–114.

[29] MAI, L., RUPPRECHT, L., ALIM, A., COSTA, P., MIGLIAVACCA, M., PIETZUCH, P., AND WOLF, A. L. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proc. 10th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2014).

[30] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the Art of Network Function Virtualization. In *Proc. 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2014).

[31] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[32] MEMBASE. Moxi – Memcached Router/Proxy, 2015. `https://github.com/membase/moxi/wiki`.

[33] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., AND WALKER, D. Composing Software-defined Networks. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[34] NETFLIX. Ribbon Wiki. `https://github.com/Netflix/ribbon/wiki`.

[35] NGINX Website. `http://nginx.org/`.

[36] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[37] OLSTON, C., REED, B., SRIVASTAVA, U., KUMAR, R., AND TOMKINS, A. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)* (2008).

[38] Open vSwitch. `http://openvswitch.org/`.

[39] PANG, R., PAXSON, V., SOMMER, R., AND PETERSON, L. Binpac: A Yacc for Writing Application Protocol Parsers. In *Proc. 6th ACM SIGCOMM Conference on Internet Measurement (IMC)* (2006).

[40] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud Scale Load Balancing. In *Proc. of ACM SIGCOMM* (2013).

[41] PRAKASH, C., LEE, J., TURNER, Y., KANG, J.-M., AKELLA, A., BANERJEE, S., CLARK, C., MA, Y., SHARMA, P., AND ZHANG, Y. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proc. of ACM SIGCOMM* (2015).

[42] RAJAGOPALAN, S., WILLIAMS, D., JAMJOOM, H., AND WARFIELD, A. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2013).

[43] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proc. USENIX Annual Technical Conference (ATC)* (2012).

[44] RIZZO, L., CARBONE, M., AND CATALLI, G. Transparent Acceleration of Software Packet Forwarding Using Netmap. In *Proc. IEEE International Conference on Computer Communications (INFOCOM)* (2012).

[45] RIZZO, L., AND LETTIERI, G. VALE, a Switched Ethernet for Virtual Machines. In *Proc. 8th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2012).

[46] SOMMER, R., WEAVER, N., AND PAXSON, V. HILTI: An Abstract Execution Environment for High-Performance Network Traffic Analysis. In *Proc. 14th ACM SIGCOMM Conference on Internet Measurement (IMC)* (2014).

[47] SONG, H. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *Proc. 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)* (2013).

[48] SOULÉ, R., BASU, S., MARANDI, P. J., PEDONE, F., KLEINBERG, R., SIRER, E. G., AND FOSTER, N. Merlin: A Language for Provisioning Network Resources. In *Proc. 10th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2014).

[49] STOENESCU, R., OLTEANU, V., POPOVICI, M., AHMED, M., MARTINS, J., BIFULCO, R., MANCO, F., HUICI, F., SMARAGDAKIS, G., HANDLEY, M., AND RAICIU, C. In-Net: In-Network Processing for the Masses. In *Proc. European Conference on Computer Systems (EuroSys)* (2015).

[50] STONE, E. Memcache Binary Protocol, 2009. `https://code.google.com/p/memcached/wiki/BinaryProtocolRevamped`.

[51] THE APACHE SOFTWARE FOUNDATION. The Apache HTTP Server Project. `http://httpd.apache.org/`.

[52] TWITTER. Twemproxy (nutcracker). `https://github.com/twitter/twemproxy`.

[53] ZHANG, W., WOOD, T., RAMAKRISHNAN, K., AND HWANG, J. SmartSwitch: Blurring the Line Between Network Infrastructure & Cloud Applications. In *Proc. 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2014).

[54] Apache Hadoop. `http://hadoop.apache.org`.

SoftFlow: A Middlebox Architecture for Open vSwitch

Ethan J. Jackson[†] Melvin Walls[¶†] Aurojit Panda[†] Justin Pettit[*]
Ben Pfaff[*] Jarno Rajahalme[*] Teemu Koponen[‡] Scott Shenker[†$]
[*]VMware, Inc. [†]UC Berkeley [‡]Styra, Inc. [$]ICSI [¶]Penn State Harrisburg

Abstract

Open vSwitch is a high-performance multi-layer virtual switch that serves as a flexible foundation for building virtualized, stateless Layer 2 and 3 network services in multi-tenant datacenters. As workloads become more sophisticated, providing tenants with virtualized middlebox services is an increasingly important and recurring theme, yet it remains difficult to integrate these stateful services efficiently into Open vSwitch and its OpenFlow forwarding model: middleboxes perform complex operations that depend on internal state and inspection of packet payloads – functionality which is impossible to express in OpenFlow. In this paper, we present SoftFlow, an extension of Open vSwitch that seamlessly integrates middlebox functionality while maintaining the familiar OpenFlow forwarding model and performing significantly better than alternative techniques for middlebox integration.

1 Introduction

With the rise of network virtualization, the primary provider of network services in virtualized clouds has migrated from the physical datacenter fabric to the hypervisor virtual switch. This trend demands virtual switches implement *virtual networks* that faithfully reproduce complex L2—L3 network topologies that were once entirely the concern of network hardware.

As network virtualization systems mature and workloads increase in sophistication and complexity, pressure continues to mount on virtual switches to provide more advanced features without sacrificing flexibility or performance. In particular, middleboxes – firewalls, NATs, load balancers, and the like – that are ubiquitous in enterprise networks [32] have begun to make their way into network virtualization systems.

Open vSwitch (OVS) – the open source virtual switch utilized by a majority of these systems – is not immune to this pressure. Its flow based forwarding model (based on OpenFlow) makes it particularly well suited to stateless L2—L3 forwarding, allowing it to achieve a high level of generality without sacrificing performance [23]. However, extending this model to middleboxes has proven difficult due to three fundamental challenges:

- Open vSwitch (and OpenFlow) models packet processing as a series of flow tables operating over packet headers. Middleboxes, on the other hand, rely on per-connection state and inspection of packet payloads that are hard to express in this model.
- In order to achieve reasonable performance, Open vSwitch uses a flow caching algorithm that depends necessarily on the stateless nature of OpenFlow to produce consistent results – packets with the exact same header must be forwarded the exact same way every single time. Middleboxes' reliance on internal state and inspection of packet payloads causes them to make *different* forwarding decisions for packets with the same header. This breaks the fundamental assumptions of the flow cache.
- Packet parsing and classification are elementary operations among all network services that long complex service chains must perform many times for a given packet. While it is feasible to integrate middleboxes with Open vSwitch using virtual machines, it's unclear how to share this work across middleboxes as Open vSwitch is able to for stateless L2—L3 OpenFlow pipelines.

In this paper we design SoftFlow, a data plane forwarding model with unified semantics for all types of packet operations. SoftFlow is an extension of Open vSwitch designed around three design principles:

Maintain the Open vSwitch forwarding model. Open vSwitch is built on OpenFlow, which has arguably helped it achieve the wide deployment it enjoys today and we see no reason to abandon it. A great deal of traditional middlebox functionality, *e.g.,* L2, L3, and ACL processing, can be implemented naturally as flows, leaving us with only a small subset of functionality that needs special processing: operations which require per-packet state and operations which inspect packet payloads. These operations can be handled with our SoftFlow extensions.

Reduce packet classifications. On general-purpose processors algorithmic packet classification is expensive. In our experience, it frequently consumes the majority of datapath CPU time and experiments in §7 indicate the same.[1] We aim to extend the benefits of Open vSwitch flow caching to middleboxes by designing for *middlebox aware* flow caching to exploit localities *across packets*, and shared packet classification *between middlebox services* to mitigate redundant computation. In doing so, we reduce the overhead of classification-heavy middleboxes like NATs and firewalls.

Increase processing locality. Running services outside of the virtual switch on separate VMs or processes provides strong isolation, both in terms of performance and memory. However, this isolation comes at a cost – performance suffers

[1]There is a vast literature on the subject of algorithmic packet classification [5, 14, 33–35]. However, we note that any of these complex algorithms performed *repeatedly* per-packet is likely to dominate processing time.

due to high virtual I/O overhead and CPU cache misses as packets traverse multiple cores. While isolation is generally assumed necessary in the NFV literature [7, 19], in systems where service implementations can be carefully vetted by a single vendor, we believe it is less critical. We choose to sacrifice isolation so we may adopt a run-to-completion model in which packets are processed by a single core, from reception, through various services, and finally to transmission. This choice leads to a factor of 2 performance boost over an NFV-style VM-based implementation.

We have implemented a prototype of SoftFlow on top of the Open vSwitch port to DPDK [9] that we are currently evaluating for inclusion in upstream Open vSwitch. In addition to the SoftFlow extensions to Open vSwitch, we have also implemented a number of service pipelines to validate the generality of the design. In what follows:

- We provide the design and implementation of an extension to Open vSwitch that supports generic L4–L7 middleboxes.
- We show how to integrate L4–L7 services with flow caching and a run-to-completion forwarding model, and provide evidence in §7 that the performance benefits of an integrated approach are as much as a factor of 2 better than VM-based middleboxes.
- We provide a design for integrating NIC hardware packet classification offload with SoftFlow, without restricting the generality of software packet processing. We show in §7 this optimization could improve forwarding rates a further 5%–90% for realistic workloads.

2 Background

While SoftFlow has broad applications, we designed it specifically to solve challenges present in the network virtualization systems [12, 16, 37]. These systems provide virtual networking to a cloud of virtual machines (or containers) running on thousands of hypervisors. Each hypervisor runs a sophisticated virtual switch, typically OVS, which is used to form an overlay of densely interconnected tunnels. In addition to the hypervisors, each system has a *gateway* which sits in between virtual networks and legacy physical networks hosting non-virtualized workloads. Gateways are typically implemented as a dedicated OVS instance running on a commodity server.

These systems have proven successful for basic stateless L2/L3 networking. However, as deployments mature and more sophisticated workloads are migrated onto virtual networks, the demand for more advanced service typical of the ubiquitous middleboxes in enterprise networks emerges. Firewalls, NATs, load-balancers, and the like are essential components of modern networks, which administrators expect to be available in their virtualized deployments.

Our primary motivation for developing SoftFlow is to support these complex middlebox services in network virtualization platforms. The system must perform well both on hypervisors, where efficiency is paramount to maximize resources available to VMs, and on gateways where performance both in terms of throughput and ability to scale to thousands of tenants, is highly valued. The peculiarities of this environment have led us to a somewhat different emphasis than is typical in the NFV literature.

- The environment is multi-tenant, but not necessarily multi-vendor. We expect most middlebox functionality to be developed specifically for Open vSwitch and carefully vetted by the community. For these reasons, strong isolation between services, while nice to have, is not critical.
- Hypervisors must balance performance and efficiency. Cloud operators are typically willing to devote a core or two to networking, but taking up the majority of hypervisor CPU resources for networking is unacceptable.
- Network virtualization systems already rely heavily on Open vSwitch and forcing them to migrate wholesale to a foreign switch based on service chains of black-box virtual machines would be, at best, burdensome.

In searching for a solution, we evaluated, attempted, and ultimately rejected two common approaches to middlebox development seen today. In the rest of the section, we discuss both in turn: the black-box model which hides middlebox complexity in virtual machines, and the "pure SDN" model which attempts to build middleboxes on top of OpenFlow.

2.1 Black Boxes

Perhaps the most common approach to building virtualized middleboxes in the literature is what we call the *black box* model. In this approach, each middlebox is a fully isolated virtual machine (or container) with a series of virtual ports connecting it to the hypervisor virtual switch. The appeal of this approach is obvious: it provides a straightforward migration path to the cloud for middlebox vendors. However, it comes with costs:

- Middlebox implementations tend to be developed and managed completely independently of the rest of the virtual network. This significantly complicates the control plane design, which must manage (at least) two completely separate systems.
- Packet transmission between cores (necessary to shuffle packets between VMs and the hypervisor) is costly. Modern systems [7, 17, 19, 23, 24, 28] mitigate this cost through shared memory, sometimes even employing zero-copy techniques. However, as we show in §7 for realistic workloads, the benefits of zero-copy I/O pale in comparison to the performance of a run-to-completion forwarding model.
- Virtual machine instantiation and management has significant overhead. In the systems that SoftFlow targets, gateways often need to run per-tenant middleboxes. For large clouds, this can add up to hundreds or thousands of instances per gateway. For instance, the Network Virtualization Platform (NVP) [16], a commercial software defined network, used Linux network namespaces (which

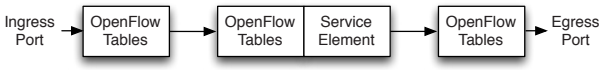


Figure 1: A conceptual model of the SoftFlow datapath. SoftFlow actions offload their classification to OpenFlow tables.

are significantly lighter than VMs) as its gateway middlebox implementation, but yet quickly encountered scale issues as management of such heavyweight appliances proved challenging. In SoftFlow, however, each middlebox is a simple function pointer in the Open vSwitch process address space – thousands can coexist trivially.

2.2 Pure OpenFlow Middleboxes

To a very limited extent, trivial middleboxes can be built on top of OpenFlow. In our quest to support middleboxes for NVP [22] we did just this with limited success. We briefly survey the pure OpenFlow middleboxes techniques we attempted, and why they fail to generalize beyond all but the simplest use cases.

When one attempts to build a pure OpenFlow middlebox, one must deal with the limitations of OpenFlow: its inability to handle stateful operations and deep packet inspection. The first attempted solution often involves a *reactive* OpenFlow model, where packets needing middlebox processing are punted to the controller for handling. It is not difficult to achieve functionally "correct" behavior with this strategy, but it is challenging to scale the approach to large networks with thousands of middleboxes forwarding millions of packets per second each.

The reactive model can be optimized by adopting a local controller co-located with each switch to handle certain types of packets. This approach scales well for services which only need to inspect the occasional packet but, beyond that, sending each packet through the Open vSwitch slow path to the local controller is both more complex and expensive than the black-box model.

Given the relatively high overhead of a local controller, a further optimization employed by Open vSwitch allows the switch *itself* to modify its own flow tables through the *learn action*. While originally designed to do L2 learning, one can implement primitive stateful services with it. For instance, a simple stateful firewall can be built by "learning a hole" (installing a new flow) for the reverse traffic of each new connection. In fact, this approach has been adopted and deployed in NVP as a primitive virtual firewall. However, it can't do true connection tracking which requires verifying the TCP state machine, implement middleboxes which require deep packet inspect, or even those that require fast changing per-packet state.

SoftFlow models the datapath as a stateless processing pipeline, implemented in OpenFlow, which forwards packets through stateful processing modules we call SoftFlow actions. These modules perform the complex processing which

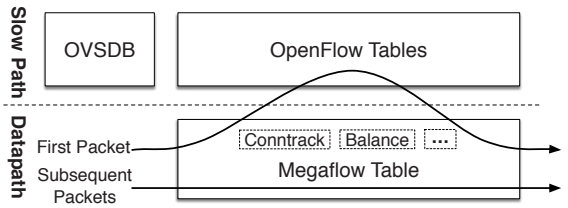


Figure 2: The components of Open vSwitch. The datapath maintains a megaflow table and the implementations of all SoftFlow actions. Packets not found in the cache are punted to the OpenFlow tables above.

OpenFlow can't support and, when finished, send packets to the next OpenFlow table for further processing. This process repeats as many times as necessary, allowing multiple stateless and stateful stages to operate on each packet.

Unlike virtual machines operating under the black-box model, SoftFlow actions can offload internal packet classifications to OpenFlow tables. For example, a controller may choose to build a firewall from an ACL table implemented in OpenFlow and a SoftFlow action which implements connection tracking. This ability to offload packet classifications has two key design benefits: First, it simplifies middleboxes by relieving them of the need to implement their own packet classification algorithms. Second, it allows middlebox packet classifications to participate in OVS flow caching along with OpenFlow tables. As a result, multiple steps from both OpenFlow tables and SoftFlow actions can be combined into a single classification per packet in the flow cache. This greatly reduces redundant packet classifications in the datapath. Figure 1 depicts this model.

3 SoftFlow Design

3.1 Flow Caching and Services

SoftFlow builds on top of Open vSwitch and inherits its flow caching architecture [25, 31] that we briefly review here. The architecture is split into two major components, a slow path and a datapath, as illustrated in Figure 2. The datapath is responsible for per-packet forwarding, which it achieves using a *megaflow* table: a giant flat (priority-less) classifier which supports bit-wise wildcarding.[2] Each rule in the megaflow table carries with it a list of actions (*e.g.,* modify headers, forward) which are performed on each packet that matches. Packets for which the datapath has no matching megaflow are punted to the slow path where the OpenFlow implementation resides. These "missed" packets are forwarded through a series of OpenFlow tables that generate a wildcard mask and action list making up a new megaflow entry. This new megaflow entry is then installed in the datapath, so that future packets can avoid the expense of a slow path traversal.

[2]Below the megaflow table there is also an exact match cache in the OVS DPDK implementation. Like Open vSwitch, SoftFlow takes advantage of this cache, but we don't describe it in detail here for brevity.

The Open vSwitch cache hierarchy described above was designed specifically to suit the needs of OpenFlow, not to support complex stateful processing necessary to implement middleboxes. In fact, as we developed SoftFlow, we identified two key limitations which must be resolved to support these workloads:

- The flow cache hierarchy is built on the assumption that each packet (as defined by its header) will be processed exactly the same way by a given set of OpenFlow tables every time. This is why a long series of OpenFlow tables can be compressed to a single packet classification in the megaflow table. SoftFlow actions, on the other hand, can make forwarding decisions based on *internal state* and *packet payload* that, from the perspective of the flow cache, are completely *nondeterministic*. This means that the flow cache must be prepared to re-classify each packet after every SoftFlow action invocation.
- The Open vSwitch slow path must know if an OpenFlow action will modify a packet, so it can take this modification into account when executing future OpenFlow lookups. While quite practical for simple OpenFlow actions, for complex SoftFlow actions this would require an implementation of the action to live in both the slow path and datapath. Furthermore, to behave correctly these two implementations would have to synchronize their internal state. While theoretically feasible, this would add a great deal of avoidable complexity and overhead.

We now consider the implementation of SoftFlow on top of Open vSwitch with these two challenges in mind. Our solutions to these problems flow naturally from three key design decisions:

Datapath Exclusive Actions. SoftFlow actions are implemented entirely in the datapath, and as a result, the slow path has no semantic understanding of SoftFlow action behavior. A pipeline of OpenFlow tables cannot continue after a SoftFlow action has executed as the slow the path doesn't know how to update the packet for the next OpenFlow table. Instead, SoftFlow actions *consume* the packets they operate on just as a VM does.

SoftFlow Stages. In order to continue a forwarding pipeline after SoftFlow actions execute, we require the controller to explicitly handle packets emerging from them at the beginning of the OpenFlow pipeline just as if they had emerged from a VM. Semantically, this requires a megaflow lookup after each action invocation (which we will optimize away shortly).

To achieve this we introduce the concept of a SoftFlow stage. Each packet entering the system is tagged with a stage id, `sf_stage`, initially set to 0. Immediately after a SoftFlow action executes, `sf_stage` increments and the packet is re-injected at the bottom of the cache hierarchy. Crucially, `sf_stage` can be matched at all layers of the cache hierarchy including the slow path, making it easy for the controller to distinguish packets emerging from SoftFlow actions from those emerging from VMs or NICs.

Metadata	Description
`sf_metadata`	Metadata register accessible by SoftFlow actions.
`sf_action`	Name of the last executed SoftFlow action.
`sf_stage`	Stage incremented after each action invocation.
`sf_coalesce`	Boolean signalling whether it is ok to coalesce.
`sf_argument`	Runtime configuration argument set by controller.

Figure 3: SoftFlow relies on a number of metadata values embedded in each packet structure. These values, and what they're for, are briefly summarized in this table..

Stage Coalescing. The Open vSwitch flow cache requires that all packets with a particular header be forwarded the exact same way every single time. This invariant holds easily for OpenFlow, but for SoftFlow actions, which can make header modifications based on internal state and packet payloads, two packets with the same header may be treated very differently. The naïve solution to this problem requires a full megaflow lookup after each SoftFlow action invocation, however, this is often unnecessary: many common middleboxes do not modify packet headers at all (*e.g.,* firewall, IDS, IPS), or if they do, modifications are deterministic given the input packet headers. On the other hand, some actions may make packet modifications on a per packet basis and really do require a megaflow lookup after each invocation.

If a SoftFlow action does not make any modifications that would require an additional megaflow lookup for that packet header, it can take advantage of a novel optimization called *stage coalescing* to avoid it. After executing, the action signals whether a new lookup is necessary by writing a boolean into the `sf_coalesce` packet metadata. If `true`, SoftFlow can skip the next megaflow lookup, instead, simply executing the actions that follow the current SoftFlow action.

The optimization is set up at megaflow installation time. If a new megaflow ends in a SoftFlow action that supports coalescing, SoftFlow tracks the packet as it flows through the stages ahead of it and *appends* the actions executed in those stages to the original megaflow's action list. Thus, when future packets hit this megaflow, they have access to all of the actions they'll need without having to execute additional megaflow lookups. Furthermore, if at any time a particular SoftFlow action needs to, it can always set `sf_coalesce` to `false` forcing a new megaflow lookup after its execution.

3.2 SoftFlow and OpenFlow

We expect a controller to configure and instantiate SoftFlow actions out of band. In our prototype implementation the set of available actions and their configuration is hard coded, but it would be simple to dynamically load new actions at runtime and configure them through OVSDB.

We used OpenFlow's vendor extension mechanism, to

add a new OpenFlow action, `softflow`, in which the controller embeds the name of the SoftFlow action to be executed, and an integer `sf_argument`.[3] This argument, used as a runtime configuration parameter, is passed to the SoftFlow action on each invocation.

Additionally, we augment OpenFlow by adding a per packet register, `sf_metadata`, which SoftFlow actions can use to exchange information. On invocation, the current value of the register is passed to the action which may be read or written it at will. After invocation future SoftFlow actions or OpenFlow tables can match on the new value and alter their behavior accordingly. In addition to the metadata register, the OpenFlow tables can also match on the `sf_stage`, and `sf_action`, the name of the most recently executed SoftFlow action.

3.3 Limitations

The tight integration of SoftFlow with the Open vSwitch data plane is not without its limitations which we discuss briefly below.

SoftFlow makes no attempt to isolate actions from each other or the rest of the platform. Since all actions share the same process address space, this implies that buggy or malicious actions can crash the switch or, worse, read from or write to other action's memory. Additionally, even if all actions are trusted and well implemented, SoftFlow provides no mechanism to fairly allocate CPU or memory resources to actions. Thus, SoftFlow is only suitable for carefully vetted trusted actions. In effect, SoftFlow stakes out an extreme position in the trade-off between strong isolation (and thus the performance cost represented by process/VM isolation) and a faster run-to-completion forwarding model.

SoftFlow is not well suited for middleboxes that rely heavily on buffering packets – specifically systems that participate in TCP connections like HTTP proxies, certain types of Intrusion Detection Systems, and some WAN optimizers. Such middleboxes require a SoftFlow action to execute the majority of processing in separate background threads, in effect preventing the run-to-completion forwarding model. While this can be implemented, after all SoftFlow actions do run arbitrary code, it would be simpler to use a virtual machine or process instead.

Finally, we note that SoftFlow makes no particular provisions for fault tolerance. In the event of failure, fail-over to a backup switch must be handled by an out of band mechanism, as is commonly done in OVS appliances today. In SoftFlow this problem is complicated by the fact that middleboxes often have internal state that must be migrated to the backup copy for correct operation. SoftFlow leaves this problem to the action implementation and makes no attempt to solve it in the framework.

[3] In future, we expect to support multiple arguments and, perhaps, more complex data types like strings. For now, however, the single argument has proven sufficient.

4 Service Design

In this section we describe the design of two software forwarding pipelines built on SoftFlow: a simple stateful firewall and a load-balancer. Later we evaluate these pipelines in §7.

4.1 Stateful Firewall

Firewalls are built of two primary components: a packet classifier that implements an Access Control List (ACL), and a connection tracker that keeps track of transport connection state. While ACLs are simple to express as OpenFlow tables, the connection tracker is difficult to express in standard OpenFlow. It's designed to allow all packets from established connections, while only allowing new connections which satisfy the ACL. Depending on the sophistication of the firewall, connection tracking may imply not only tracking TCP connection state, but also validating TCP sequence numbers.

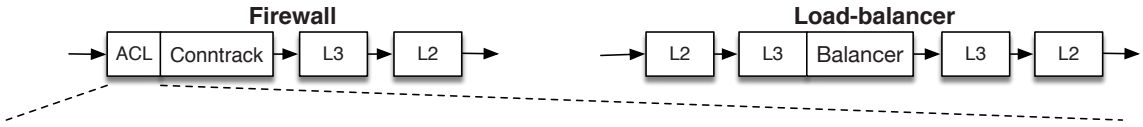
Our firewall implements its ACL table in OpenFlow, which works cooperatively with a SoftFlow `conntrack` action responsible for connection tracking. In our prototype implementation `conntrack`'s internals are based on a connection tracker designed for OVS-DPDK [2] that is itself based on the BSD firewall.

Our firewall pipeline begins when packets are forwarded to the ACL OpenFlow table. If any ACLs successfully match a packet, it passes to the `conntrack` action with `sf_argument` 1. This `sf_argument` value indicates to the action that the packet successfully matched an "allow" ACL and, if necessary, a new connection entry may be created. The table also has a low-priority default rule for those packets that don't match any ACL. These packets are still passed to the SoftFlow action, but with `sf_argument` value 0, indicating that the packets failed to match the ACL table, and should only be forwarded if they belong to an existing connection.

The `conntrack` action executes and, in the process, writes to the packet's `sf_metadata` the value `1` if the packet is allowed, or `0` if it should be dropped. When finished, control returns back to the datapath which increments `sf_stage` and passes the packet back into the OpenFlow tables.

To avoid conflicts with other rules, the OpenFlow table matches on `sf_stage=1` and `sf_action=conntrack` to catch packets received from the connection tracker. These rules either drop failures or instruct successes to proceed to the next table. Finally, the packet traverses L3 and L2 tables which determine the appropriate output port. Figure 4 depicts the resulting SoftFlow firewall pipeline as well as its first flow table, both responsible for processing packets incoming from the NIC and from the conntrack service.

Packet walkthrough. A packet entering the datapath is first checked against the megaflow cache. If empty, the packet enters the OpenFlow slow path, which is responsible for compiling a datapath action list and a wildcard mask for a new megaflow cache entry. The packet begins in the initial




```
priority=100 sf_stage=1 sf_action=conntrack sf_metadata=1 -> goto L3 table
priority=100 sf_stage=1 sf_action=conntrack sf_metadata=0 -> drop
priority=100 sf_stage=0 ip_dst=10.0.0.3 tcp_src=80        -> softflow:conntrack:1
priority=0   sf_stage=0                                    -> softflow:conntrack:0
```

Figure 4: A simple pipeline for a firewall (left) and load-balancer (right). Below, a simple firewall ACL table that allows 10.0.0.3 to initiate TCP connections on port 80 and allows all return traffic; all other traffic is dropped. The first two flow entries match traffic coming back from the conntrack service. The contents of the firewall L2 and L3 forwarding tables, as well as LB forwarding tables, are omitted for brevity.

OpenFlow table, a packet classification is performed, and a matching ACL is chosen.

The OpenFlow action corresponding to the matching ACL indicates that the conntrack service should be invoked. However, the slow path *does not execute* the service itself as the actual implementation lives entirely in datapath. Instead, the slow path simply appends the SoftFlow action, with some additional metadata, to the action list gathered so far and passes this list to the datapath as a new megaflow cache entry.

The datapath then executes the action list. The final item in the list is a reference to the SoftFlow action along with a function pointer and the `sf_argument` necessary to invoke it. The datapath executes the action by making a simple function call, passing along the packet and `sf_argument`.

The action executes, runs the packet through its connection tracking logic, and sets `sf_metadata` to `1` meaning "pass". Since "pass" is the typical result, the action also sets `true` to `sf_coalesce` meaning this packet may be coalesced. Control returns to the SoftFlow datapath which increments `sf_stage`, and sets `sf_action` to "conntrack".

At this point, a normal Open vSwitch would install the rule in its megaflow cache. However, since `sf_coalesce` is `true`, coalescing is enabled for this packet. Therefore it is passed back to the slow path once again.

Receiving the packet for the second time, the slow path again runs the packet through the standard OpenFlow pipeline starting at the first table. Since `sf_stage` is `1`, `sf_action` is "conntrack", and `sf_metadata` is `1`, the first rule in the figure matches and the packet proceeds to the L2/L3 tables.

Again, the slow path assembles a mask and action list, and hands them to the datapath. The cache executes the action list just as before, then *appends* the results to the action list collected from the previous slow path invocation. By doing so, the flow cache coalesces what would have been two stages into a single megaflow, effectively reducing the number of megaflow lookups for future packets to one.

Since no more SoftFlow actions are executed, the process terminates and the cache entry is finally installed. Future packets entering the system perform a megaflow lookup to find this newly installed megaflow and execute the action list without the overhead of a miss or several packet classifications. In the common case, this megaflow lookup will be handled by the exact-match cache, requiring just a single hash table lookup in addition to the cost of connection tracking.

4.2 Load Balancer

We now detail a load balancer (evaluated in §7) which balances TCP connections among a configurable number of backends. The load balancer is constructed from two pieces: an abstract `balancer` SoftFlow action and a controller-provided OpenFlow table which acts on the action's decisions. The load balancer assigns each connection to one of eight buckets and writes its choice to `sf_metadata`.

Just like the firewall, after the `balancer` action is executed, the packet is passed back to the OpenFlow table. In this table, a set of rules is installed that map each `sf_metadata` value to a backend host. The OpenFlow table transforms the packet accordingly (by setting the appropriate destination MAC and IP) before forwarding it to the next table for an L3 lookup, L2 lookup, and eventual transmission.

Packet walkthrough. Similar to the firewall, packets entering the datapath traverse the cache hierarchy eventually ending up in the slow path. The slow path compiles a new megaflow for installation that references the `balancer`. The datapath executes the action list, including `balancer` which takes the following steps:

- Assuming the packet belongs to a new connection, the `balancer` implementation creates a connection identifier for the packet, and assigns it to the least loaded of eight[4] available buckets.
- A map is updated noting which bucket this connection is assigned to so that future packets are forwarded consistently.
- The bucket's byte count is incremented with the packet's size, so that future packets can make appropriate balancing decisions.
- The packet's `sf_metadata` register is updated with the

[4]Note that eight was chosen arbitrarily, any small number would do.

ID of the chosen bucket.

- Finally, the balancer returns `false` to the SoftFlow datapath indicating that the result cannot be coalesced.

From the perspective of OpenFlow, the load balancer's behavior is *nondeterministic* regarding L2–L4 input headers. Its choice of bucket is based on internally maintained load information that OpenFlow does not have access to. Therefore, `balancer` always returns `false` to the datapath indicating that coalescing should never occur. For our packet, this means the megaflow is installed immediately, just as it would have been with standard Open vSwitch.

After the megaflow is installed, the packet is re-injected at the bottom of the cache hierarchy where it eventually finds its way back to the slow path OpenFlow tables. The packet traverses the OpenFlow tables, various transformations are made, and again a new megaflow is created. This time the megaflow is scoped to `sf_stage=1`, so that future packets emerging from the `balancer` action that are assigned to the same bucket, can skip the slow path entirely and need only perform a packet classification in the datapath. Once enough traffic has been processed to set up a megaflow per bucket, future packets can be forwarded without involving the slow path at all, but at the cost of two classifications: one to decide that the packet requires load-balancing and one to decide what actions to take based on the load-balancer's decision.

5 Implementation

In this section, we discuss details of our prototype SoftFlow implementation built on the Open vSwitch port to DPDK, OVS DPDK. In our prototype, a configurable number of cores are dedicated to forwarding, each of which runs a dedicated polling thread. Each NIC has a dedicated receive queue per thread over which packets are load balanced using a simple hash over the 5-tuple. Polling threads process packets using a run-to-completion model, meaning that the core which receives a packet sees it through all the way to transmission, never handing it to another thread for processing. Our prototype implementation is built on top of OVS 2.4 with a series of patches totalling approximately 1800 lines of code not including SoftFlow action implementations.

5.1 Service API

In our prototype, service implementations are directly compiled and linked to the Open vSwitch process, and statically configured for simplicity. This prototype mechanism could be replaced with a dynamic library loading of the services through a stable ABI with configuration through OVSDB.

SoftFlow maintains a global list of all SoftFlow services which, today, is simply hard-coded in the source (though future versions will allow it to be dynamically loaded at runtime). As Open vSwitch boots, it initializes each service using their internal initialization functions, and makes the services available to OpenFlow and the datapath for use.

In SoftFlow, services can form groups for their internal state sharing purposes. Thus, the actual service registration takes place at the group level: services are constructed as groups of actions and the service group is registered through a `sf_group` structure. In it, references to the service element specific initialization structures are provided.

Each service element provides a `sf_service` structure which specifies an instance initialization function (provided with a reference to the group) as well as a function for the SoftFlow datapath to invoke to process packets:

```
struct sf_service {
  const char *name;

  /* Construct and return a new instance. */
  void *(*init)(struct sf_group *group);

  /* Process a batch of packets. */
  void (*ingress)(struct pkt **pkts, size_t n,
                  void *instance);
};
```

This is the *entire* interface an action has to implement. Afterward, OpenFlow entries can refer to the action and the SoftFlow datapath can invoke the ingress function to execute batches of packets. The function's arguments are a batch of packets and an "instance" pointer to service internal state. Packet metadata is held within the packet construct.

5.2 Datapath Integration

While at the OpenFlow protocol level SoftFlow actions are quite similar to standard OpenFlow actions, their actual implementation is completely different from these simple procedures.

At OpenFlow rule install time, the SoftFlow action is translated to an internal representation which directly holds a pointer to the `ingress()` function and the metadata necessary to call it. When the slow path encounters this action, it in turn converts the packet to a flow cache specific internal representation which then contains the necessary information to invoke the action; that is, ingress function, instance pointer, and `sf_argument`. Thus, the rest of the flow cache can remain unaware of the intricacies of SoftFlow action invocation, and instead simply call a function pointer.

Coalescing. When a megaflow cache miss occurs in the standard Open vSwitch implementation, the packet is passed to the slow path, a new megaflow entry is returned, the entry is installed, and the packet is forwarded based on this new entry. In SoftFlow, however, this simple process must be enhanced to support stage coalescing. After receiving a new megaflow from the slow path, if the last action is a SoftFlow service, a different procedure is executed:

- The action list associated with the new megaflow is executed, including the final SoftFlow action.
- The SoftFlow action updates the `sf_metadata` and (let's assume for this example) sets `sf_coalesce` to `true`.

- The `sf_stage` increments, and `sf_action` updates.
- The packet is passed back to the slow path which generates a new megaflow.
- The new megaflow is merged with the previous megaflow, and its actions are appended to the currently accumulating list.
- The process repeats until a megaflow returns without a SoftFlow action, or a loop detector triggers.

5.3 Actions

In addition to the SoftFlow datapath prototype, we developed several SoftFlow actions in an effort to evaluate both the efficacy of our design as well as to understand its implications to a service developer. As shortly discussed in the next section, these services range from a trivial packet counter, to an AES payload transcoder, connection tracker, and load balancer. There are several unique peculiarities to SoftFlow action development:

- SoftFlow instances are fine-grained and specific to a particular action configuration. A SoftFlow load-balancer, for example, can assume an incoming packet is TCP, is heading to a particular Virtual IP address, and runs over an HTTP port, because the OpenFlow flow table will guarantee it receives such packets. As a result, a service can be broken into smaller, more manageable, modules with simpler internals.
- SoftFlow services are based on the Open vSwitch code, and inherit the internal packet representation used throughout Open vSwitch. This internal representation contains pointers to the L2, L3, and L4 header offsets, saving SoftFlow actions the expense of re-parsing each packet.
- SoftFlow provides packets to service instances in batches. This allows implementations to utilize *prefetching* as per the DPDK guidelines: first prefetching the necessary internal state for a batch of packets into CPU cache, and only after that processing packets using that internal state. Such "staged" operation slightly complicates service implementations but is effective in reducing latency due to CPU cache misses.

6 Hardware Classification Offload

NIC vendors have a long history of introducing hardware acceleration techniques, with mixed success. TCP checksum and segmentation offload have proven useful, but a wide range of other NIC functions have been commercial failures. We contend this is due to an attempt to offload too much functionality. For instance, SR-IOV bypasses the virtual switch within hypervisors. While such an approach results in good performance, hardware is inflexible, has limited functionality, and has long update cycles that prevent it from adapting to new use cases. In this section, we discuss how, working cooperatively with the hardware, OVS can leverage classification offloads, and SoftFlow can take advantage of these offloads despite having *general* middlebox actions. This approach allows SoftFlow to benefit from the performance of hardware offload while maintaining the *generality* and *flexibility* of software forwarding.

Current commercially available high-end NICs already have a TCAM on-board, but its functionality is limited to QoS and Receive Side Scaling (RSS). For these use cases, the TCAM classification directs packets to different priority queues based on incoming L2–L4 headers [10]. Next-generation NICs including Intel's Boulder Rapids and Broadcom's BCM57300 NetXtreme C-Series [20] expand on this capability capability by modeling the TCAM as a generic OpenFlow-like switch.

While providing a programmable OpenFlow switch on a NIC partially alleviates flexibility concerns, a complete offload of switch functionality to the NIC presents challenges:

- **TCAM capacity.** TCAMs on NICs are quite limited in size and support only a fixed set of protocol header fields. This implies the maximum size of an offloaded OpenFlow table may be too small for practical classification offloading. Current NIC TCAMs have 16k entries or less, depending on the number of fields matched on.
- **Stateless operations only.** Executing OpenFlow actions on the NIC is enough for stateless L2–L4 operations, but stateful L4–L7 cannot be offloaded.

In SoftFlow, instead of offloading *all* classification to the NIC, we instead use the TCAM as a hardware flow cache *assisting* the software flow cache. This allows SoftFlow to offload the most active megaflows and leave the rest to software making TCAM limitations less of a concern. Similarly, since actions are performed by the CPU, limits on the actions available on the NIC are of no consequence.

To accelerate classification, we assign each megaflow a unique ID before pushing it into the NIC. Furthermore, we configure the NIC actions associated with each megaflow to write this unique ID into the metadata of each matching packet. If the NIC finds a match, software classification can be skipped, allowing SoftFlow to proceed to executing OpenFlow actions and SoftFlow services. Similarly, to work around matching limitations, SoftFlow can offload matches over the limited headers supported by the NIC, leaving the rest for software. We do not discuss this here further because it is explored deeply in SAX-PAC [14] and we expect future L2–L4 headers to be relatively well supported through more programmable packet parsing capabilities in future NICs.

While in general guaranteeing consistency of TCAM table updates is a potentially complicated topic [38], the priority-less eventually consistent design of the megaflow cache simplifies the problem. On reception of a packet with a TCAM hint from the NIC, we simply verify that the indicated megaflow still exists and the packet does, indeed, match it. Thus if the megaflow table changes in a way inconsistent with the NIC's TCAM, affected packets will simply fall back to software classification until the NIC is updated. Similarly, if a megaflow matches on fields

unsupported by the hardware, the discrepancy is resolved in software for misclassified packets.

Finally, while the offloading remains completely transparent to the service implementations – after all it is completely the responsibility of the SoftFlow datapath – the converse is not true. Services that cannot be coalesced require re-classification after execution. In SoftFlow these secondary classifications are not done at ingress, and therefore must be done in software. That is, SoftFlow cannot offload all packet classification to the NIC

7 Evaluation

In this section, we evaluate the benefits of SoftFlow's run-to-completion architecture, flow caching, and stage coalescing for workloads requiring complex middlebox services.

7.1 Test Environment

Our testbed has two identical servers each running a pair of 10-core Intel Haswell 2.6GHz CPUs with hyper-threading disabled, 25MB of L3 cache and 128GB RAM. Each server has an Intel 10Gb NIC with two ports for a total forwarding capacity of 20Gbps. The ports of each server are patched directly into the other with no switches or routers mediating them. One server is configured as a packet generator running Pktgen-DPDK [26], the other runs our SoftFlow prototype. In all tests, the SoftFlow test server receives packets, forwards them through a pipeline of OpenFlow tables and SoftFlow services, and forwards them back out the port on which they were received.

Except where noted, all experiments are run with a single SoftFlow core (as would be typical on a hypervisor), with coalescing enabled (for the actions that can take advantage of it), the TCAM simulator disabled (as initially most deployments won't have TCAM accelerated NICs), and a 100% megaflow cache hit rate after a brief ramp up period.

Packet traces. We evaluate three traces, which the traffic generator replays repeatedly at line rate on both ports for a total throughput of 20Gbps. The first trace, T1, was collected from a software network virtualization gateway appliance running Open vSwitch that was deployed in a private, production, multi-tenant datacenter servicing approximately one thousand hypervisors. The trace is 44 seconds long, contains 4.6 million packets, with an average packet size of 937 bytes, and average transmission rate of 795Mbps. We replay the trace at line rate and, on each replay, we replace the L4 source and destination ports with a consistent hash of their previous values to defeat the exact match flow cache. The trace contains 11k IP addresses participating in 100k distinct TCP conversations and 75k distinct UDP conversations. The maximum conversation length in our trace is 99k packets, the mean is 42 packets, and the median is 10 packets.

To further stress the prototype, we stripped the payload of T1 to simulate traffic with maximum packet size of 64 bytes. In our tests, the resulting packet trace is called "T2". In addition, to simulate ideal conditions, we generated a synthetic trace called "T3" consisting of a few long-lived, high volume connections– specifically, 32 transport connections, each in turn sending a burst of 256 packets of 64 bytes each.

7.2 Pipelines

We implemented four prototype SoftFlow pipelines to emulate use cases typical of network virtualization systems.

Pipeline A. Our first pipeline mimics a network virtualization gateway as described in [16]. It consists of four stages starting with 500 OpenFlow rules that match randomly generated Ethernet destination addresses. From there, it continues to an L3 table whose rules match 500 randomly generated IP prefixes. (Random flow tables like these are a worst case for Open vSwitch because they result in the most specific flow masks used in the flow cache [25].) Next is a SoftFlow firewall consisting of 500 ACLs and a SoftFlow `conntrack` action, as described in §4. The firewall is quite sophisticated as it maintains per-connection state, tracks TCP sequence numbers, and supports multiple protocols beyond TCP. The ACLs are a randomly chosen subset of a cloud provider's production firewall rule set. The firewall is followed by an L2 table, after which packets egress.

Pipeline B. The second pipeline demonstrates SoftFlow performance for a complex service chain of actions. The first stages are identical to Pipeline A: an L2/L3 lookup and stateful firewall. Following the firewall, we execute a content transcoder which encrypts the transport payload of each packet using AES-128 in CBC mode (similar to IPsec [13]). Like the firewall and load balancer, the content transcoder is implemented as a SoftFlow action. Finally, a stateless OpenFlow NAT moves the IP destination of the packets into the 10.0.0.0/8 prefix.

Pipeline C. The third pipeline demonstrates the additional cost of making forwarding decisions based on internal state. It is identical to Pipeline A except an implementation of the load balancer described in §4 is inserted just after the firewall. As discussed earlier, the load balancer defeats coalescing and therefore requires a megaflow lookup after each invocation. (In contrast, Pipelines A and B use coalescing.)

Pipeline D. Our final pipeline, D, is a configurable number of "no-op" SoftFlow actions designed to isolate action invocation overhead. For brevity, we only present results for this pipeline in a couple of cases that are particularly interesting.

7.3 Measurements

Run-to-Completion. A common approach to virtualizing middleboxes, both in practice and in the literature [7, 19, 24] is to allocate a dedicated virtual machine to each middlebox. This approach provides strong isolation between middleboxes, and a simple migration path from legacy implementations, but as we show in Figure 5, it comes at significant performance cost.

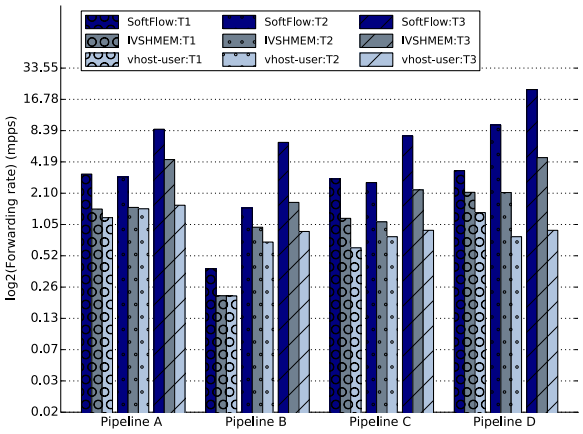


Figure 5: Comparison of forwarding rates (on a log scale) between SoftFlow and KVM virtual machines running one of two virtual NIC implementations: IVSHMEM and vhost-user.

We evaluate each of our test pipelines against equivalent virtual machine implementations in which each middlebox (*i.e.,* a SoftFlow action and its associated OpenFlow tables) is allocated its own dedicated SoftFlow VM: one for Pipeline A's firewall, two for Pipeline B (firewall and AES), two for Pipeline C (firewall and load balancer), and one for Pipeline D's `noop` action. The stateless L2 and L3 lookups not associated with a middlebox are implemented in the hypervisor vswitch (running the same SoftFlow implementation but without any SoftFlow actions). The hypervisor vswitch and the virtual machines are each allocated a dedicated CPU core for forwarding so, in total, Pipeline A runs 2 cores, Pipeline B runs 3, and Pipeline C runs 3.

We compare the VM implementation to a pure SoftFlow stack configured using run-to-completion forwarding. For each pipeline, the SoftFlow switch is allocated the same number of cores as the equivalent VM test so that performance comparisons are fair.

Conventional wisdom suggests that forwarding performance is determined by the virtual NIC. Therefore, we compare SoftFlow against two competitive DPDK virtual NIC drivers, IVSHMEM [11] and vhost-user [36]:

- *IVSHMEM*: a zero-copy implementation not dissimilar from the approach taken by NetVM [7]. This implementation relies on a shared memory region between hypervisor and guest over which pointers to packets are transferred. Note that the putative advantage of this approach is speed *at the cost of isolation* as virtual machines have read and write access to all packets on the hypervisor.
- *vhost-user*: the virtual NIC implementation officially recommended by Open vSwitch was developed in response to the limitations of IVSHMEM. The approach is similar to that taken by ClickOS [19] – packets are copied into and out of a shared memory region in the virtual machine address space. The overhead of each packet IO is the same as IVSHMEM except for an additional packet copy. In return for this copy, vhost-user achieves true isolation making it a suitable choice for untrusted virtual machines.

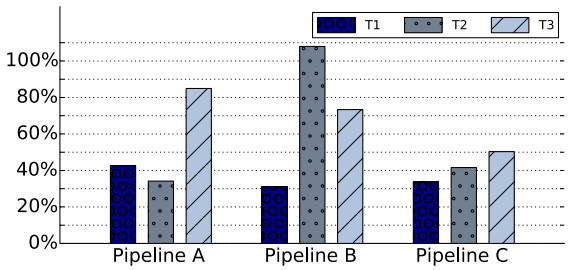


Figure 6: Percent improvement in single core forwarding throughput with stage coalescing enabled.

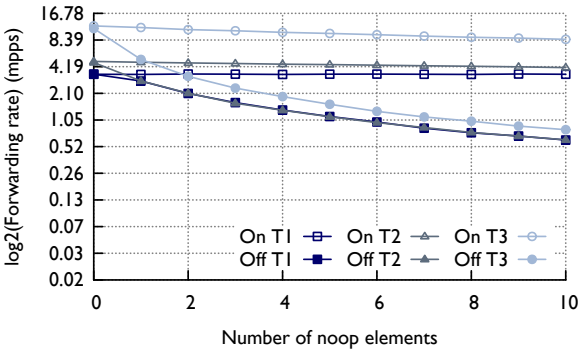


Figure 7: Single core forwarding rates (mpps) as a function of the number of `noop` service elements in Pipeline D. For each trace we test with coalescing enabled and disabled.

These two approaches represent a baseline similar to VM based NFV proposals like NetVM and ClickOS with their highly efficient virtual NIC implementations. IVSHMEM and vhost-user, are analogous to these approaches, but using OVS, thus allowing our results to isolate the cost of VM traversal versus the SoftFlow run-to-completion forwarding model.

As shown in Figure 5, SoftFlow consistently outperforms virtual machines by *2x* or more. Also, interestingly, the difference between IVSHMEM and vhost-user is less than we expected in many cases. For these cases, we believe the cost of VM traversal, (CPU cache misses, packet re-parsing, additional classification) outweights packet copies.

Stage coalescing. Figure 6 shows the percent improvement in forwarding rate caused by enabling classification coalescing for each pipeline and traffic pattern. While all cases benefit somewhat, the degree of the benefit is highly dependent on the pipeline and traffic pattern. Pipeline B, for instance, does AES encryption on the payload which mitigates the benefits of coalescing for T1's large packets, while for the small packets of T2 and T3, encryption is less dominant allowing coalescing to help more. Also, Pipeline C implements the `balancer` action which never coalesces packets, limiting the overall benefit. We also note that we

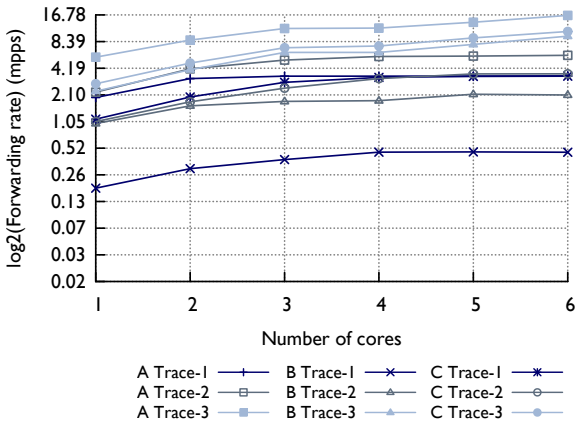


Figure 8: Performance as number of CPUs increases.

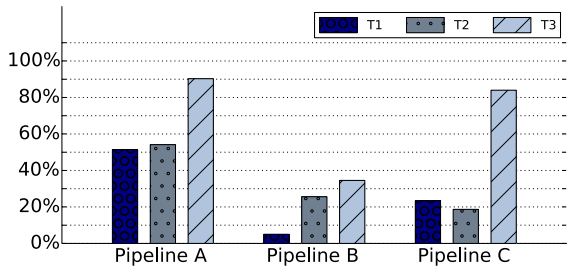


Figure 9: Percent improvement in single core forwarding throughput with TCAM offload and full stage coalescing enabled (see Figure 6).

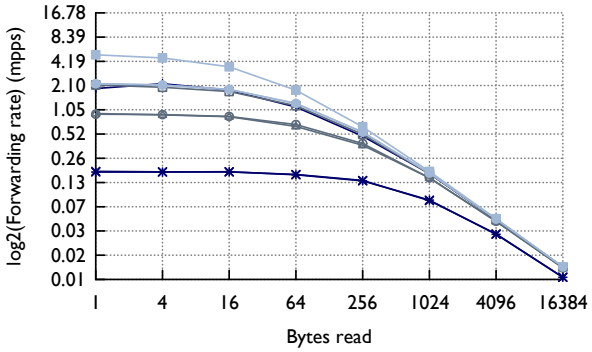


Figure 10: Single core forwarding rates when a service reads increasingly many random bytes of a 1GB block. See Figure 8 for the legend.

measured performance of coalescing as decreasing fractions of traffic were allowed to benefit, we've omitted the graph for brevity, but report that the benefit scales smoothly.

We additionally measure the effect of stage coalescing on Pipeline D. Figure 7 shows the packets per second forwarded with coalescing both enabled and disabled as the number of `noop` actions in the pipeline increases. Note that even for this trivial pipeline, traversing the cache hierarchy has real cost that can be avoided with stage coalescing.

Hardware offload. We do not yet have an engineering sample of a NIC with a programmable TCAM, so instead we emulated 1k TCAM entries in our existing testbed. Megaflow cache sizes witnessed with the above tests were rationale for this TCAM size: for instance, Pipelines A, B, and C resulted in 258, 21,335 and 256 active megaflows in coalesced tests with Trace 2. Tests in hypervisor virtual switch environments similarly suggest that a few hundred megaflows is sufficient for high hit rates [25]. We preprocessed the traces, encoding a rule ID for each connection in the IP ID field. Then we forwarded the packets through the prototype, and checked the field to choose a matching megaflow, thus allowing us to skip the megaflow classification. Figure 9 shows the percent improvement this feature provides. Note that the test was performed with stage coalescing enabled, so this benefit is on top of what coalescing can provide. The benefit depends highly on the pipeline and traffic pattern, ranging from 5% for Pipeline B (AES on large packets dominates), to 90% for the synthetic traces. We also measured fractional TCAM offload rates, and found the benefit to scale smoothly as offload rates approach 100%. We omit the graph for brevity.

Multi-core parallelism. The Open vSwitch datapath is designed to scale easily as NICs load-balance traffic across CPU cores, sharing little state. For this reason, and our focus on hypervisors where low CPU utilization is paramount, the majority of this evaluation has focused on single-core performance. However, in Figure 8, we did measure the affect of additional CPUs on performance. Both Pipeline A T1 and Pipeline C T1 quickly hit the limits of our 20Gbps test bed. The rest of the lines scale well, though sometimes hitting points of diminishing returns due to inefficiencies within the SoftFlow actions, or Open vSwitch itself.

Action Complexity. Packet processing on modern general-purpose processors is highly sensitive to memory access patterns. In an attempt to quantify the performance of complex SoftFlow actions which require traversing large complex data structures, we injected additional, synthetic memory load to the pipelines by introducing a service element that does nothing but make a series of random prefetched memory accesses. In Figure 10 we see how, regardless of successful flow caching, the throughput quickly drops as more random memory locations are accessed per packet.

Cache miss rate. Maintaining high flow cache hit rates is also critical for optimal throughput. While effective caching is the task of Open vSwitch flow cache architecture, we tested the overall impact of lowering cache hit rates for SoftFlow pipelines with L4–L7 services by synthetically introducing flow cache misses. Figure 11 validates the criticality of maintaining high cache hit rates for overall forwarding throughput.

CPU. Finally, we analyzed how the breakdown of CPU usage evolves if classification coalescing and TCAM

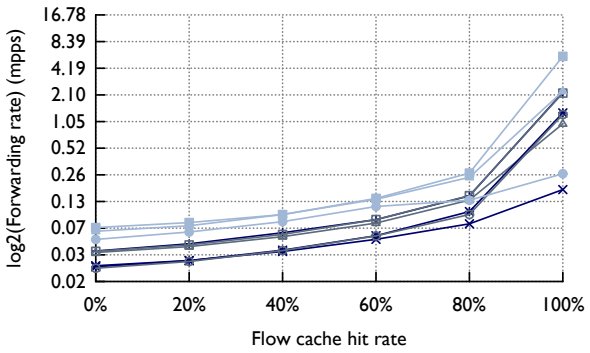


Figure 11: Single core forwarding rates as flow cache hit rate improves. At 0% the slow path handles all packets. See Figure 8 for the legend.

offloading are enabled with the Pipeline A and T2. In Figure 12, we see how both stage coalescing and hardware offloading reduce the portion of CPU used for megaflow lookup (the "Megaflow" column in the table), thus leaving more CPU resources for service execution. This naturally implies that the share of packet parsing and I/O grows.

8 Related Work

We now consider the related work in addition to the high-level approaches discussed in §2. First, we note that the interest in software forwarding was revived by the low-level hardware and software I/O optimizations that were identified to be necessary to substantially improve the ability of x86 to forward small packets at high rates [3, 8, 9, 27]. SoftFlow builds on the results of these efforts.

NFV and the requirements of datacenter workloads calling for high throughput networking capacity (both in packet rates and volume) have resulted in many privately and publicly documented optimizations in the virtual NIC I/O interface hypervisors provide for the VMs [7, 19, 28, 29]. Contrary to some of these efforts being used to integrate services as VMs, SoftFlow builds on tighter integration of packet processing across different service elements. As a result, while an unfair and incomplete comparison, SoftFlow is often able to reach line rate of 10Gbps even with one or two CPU cores, whereas VM based approaches use all the cores a server has to offer to accomplish the same with multi-stage pipelines [7, 19].

Perhaps the most closely related work is, therefore, the recent proposals arguing for the benefits of *vertical* integration of protocol stacks in userspace, to both improve performance [18] and accelerate innovation [6]. In SoftFlow, we take the integration a step further and demonstrate that also *horizontal* integration across all packet processing elements (L2–L7) can improve the overall performance.

An important alternative to the SoftFlow forwarding model was first proposed by Click [15], and adapted by ClickOS [19] for NFV workloads. This approach models switch processing as a graph of composable packet

	Megaflow	Services	Parsing	I/O
Base	57%	8%	6%	3%
Coalesce	44%	15%	5%	11%
TCAM	18%	28%	9%	20%

Figure 12: Approximate CPU utilization of various components for Pipeline A T2 with coalescing off, coalescing on, and coalescing combined with TCAM offload. Functions which could not be accounted to a particular column (*e.g.,* `memcmp()`) or which took less than 2% of CPU are omitted from the table.

processing elements. SoftFlow actions are analogous to Click elements, yet they operate in a very different context. SoftFlow only uses these actions for stateful processing, falling back to OpenFlow style match-action tables for everything else. This allows SoftFlow to take advantage of global flow caching, and classification coalescing, both of which would be impossible in the Click forwarding model. For packet classification heavy workloads, like network virtualization, this makes SoftFlow an ideal fit.

Open vSwitch has taken some steps to add common middlebox functionality, most notably with the addition of the `conntrack` action that allows packets to access a connection tracker within the datapath. Due to the lack of a framework like SoftFlow, this work requires significant manual development effort for each new action touching all parts of the OVS code base. Additionally, these efforts do not take advantage of SoftFlow classification coalescing and, thus, require unnecessary megaflow lookups.

There has been some work offloading the entire Open vSwitch datapath to an NPU accelerated NIC [21]. This design does not work in cooperation with middleboxes as SoftFlow does. On the other hand, the SoftFlow hardware offload design only offloads expensive packet classification, leaving complex action processing to the CPU.

While, there has been significant work building integrated middleboxes on commodity systems [1, 4, 24, 30], we are the first system to specifically target the vast deployment of OpenFlow based network virtualization platforms with an architecture that's practically deploy-able, and tightly integrated with stateless L2–L3 services.

9 Conclusion

In this paper we described the design of SoftFlow, an extension of Open vSwitch designed to bring tightly integrated middleboxes to network virtualization platforms. Contrary to traditional software datapath designs, SoftFlow integrates network services tightly together to facilitate pervasive flow caching, removal of redundant packet classifications through stage coalescing, and the use of hardware classification offloads. These advantages coupled with SoftFlow's run-to-completion forwarding model allow it to significantly outperform virtual machine based alternatives, while being a better fit for existing Open vSwitch deployments.

References

[1] J. W. Anderson, R. Braud, R. Kapoor, G. Porter, and A. Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. of ANCS*, 2012.

[2] Daniele Di Proietto. `https://github.com/ddiproietto/ovs/tree/userconntrack_20150908`.

[3] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. of SOSP*, 2009.

[4] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-Defined Middlebox Networking. In *Proc. of HotNets*, 2012.

[5] P. Gupta and N. McKeown. Packet Classification on Multiple Fields. In *Proc. of SIGCOMM*, August 1999.

[6] M. Honda, F. Huici, C. Raiciu, J. Araújo, and L. Rizzo. Rekindling Network Protocol Innovation with User-level Stacks. *SIGCOMM CCR*, 44(2):52–58, 2014.

[7] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI*, 2014.

[8] Intel Data Direct I/O Technology, 2013. `http://www.intel.com/content/www/us/en/io/direct-data-i-o.html`.

[9] Intel DPDK: Data Plane Development Kit, 2013. `http://dpdk.org`.

[10] Intel Ethernet Flow Director. `http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html`, September 2015.

[11] IVSHMEM Libarary. `http://dpdk.org/doc/guides/prog_guide/ivshmem_lib.html`.

[12] Justin Pettit and Ben Pfaff and Chris Wright and Madhu Venugopal. OVN, Bringing Native Virtual Networking to OVS.

[13] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, IETF, December 2005.

[14] K. Kogan, S. Nikolenko, O. Rottenstreich, W. Culhane, and P. Eugster. SAX-PAC (Scalable And eXpressive PAcket Classification). In *Proc. of SIGCOMM*, 2014.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[16] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Proc. of NSDI*, 2014.

[17] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proc. of ASPLOS*, 2013.

[18] I. Marinos, R. N. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proc. of HotNets*, 2013.

[19] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of NSDI*, 2014.

[20] `https://www.broadcom.com/press/release.php?id=s923886`.

[21] R. Neugebauer. Selective and Transparent Acceleration of OpenFlow Switches (Netronome Whitepaper), July 2014. `https://netronome.com/wp-content/uploads/2014/07/Netronome-Selective-and-Transparent-Acceleration-of-OpenFlow-Switches-Whitepaper_4-13.pdf`.

[22] Network Virtualization Platform. `http://www.nicira.com/en/network-virtualization-platform`.

[23] Open vSwitch – An Open Virtual Switch. `http://www.openvswitch.org`, September 2015.

[24] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proc. Symposium on Operating Systems Principles*, October 2015.

[25] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *Proc. of NSDI*, 2015.

[26] Pktgen Traffic Generator Using DPDK. `https://github.com/Pktgen/Pktgen-DPDK/`, September 2015.

[27] L. Rizzo. Netmap: a Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC*, June 2012.

[28] L. Rizzo and G. Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proc. of CoNEXT*, 2012.

[29] L. Rizzo, G. Lettieri, and V. Maffione. Speeding Up Packet I/O in Virtual Machines. In *Proc. of ANCS*, 2013.

[30] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and implementation of a consolidated middlebox architecture. In *Proc. of NSDI*, 2012.

[31] N. Shelly, E. Jackson, T. Koponen, N. McKeown, and J. Rajahalme. Flow Caching for High Entropy Packet Fields. In *Proc. of HotSDN*, 2014.

[32] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. of SIGCOMM*, 2012.

[33] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *Proc. of SIGCOMM*, August 2003.

[34] V. Srinivasan, S. Suri, and G. Varghese. Packet Classification Using Tuple Space Search. In *Proc. of SIGCOMM*, 1999.

[35] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proc. of SIGCOMM*, August 2010.

[36] Vhost Libarary. `http://dpdk.org/doc/guides/prog_guide/vhost_lib.html`.

[37] VMware NSX. `http://www.vmware.com/products/nsx`, September 2015.

[38] Z. Wang, H. Che, M. Kumar, and S. K. Das. CoPTUA: Consistent Policy Table Update Algorithm for TCAM Without Locking. *IEEE Trans. Comput.*, 53(12), December 2004.

Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers

Guo Chen[1,2], Yuanwei Lu[3,2], Yuan Meng[1], Bojie Li[3,2], Kun Tan[2], Dan Pei[1*], Peng Cheng[2], Layong (Larry) Luo[2], Yongqiang Xiong[2], Xiaoliang Wang[4], and Youjian Zhao[1]

[1]*Tsinghua National Laboratory for Information Science and Technology, Tsinghua University,*
[2]*Microsoft Research,* [3]*University of Science & Technology of China,* [4]*Nanjing University*

Abstract

To achieve low TCP flow completion time (FCT) in data center networks (DCNs), it is critical and challenging to rapidly recover loss without adding extra congestion. Therefore, in this paper we propose a novel loss recovery approach FUSO that exploits multi-path diversity in DCN for transport loss recovery. In FUSO, when a multi-path transport sender suspects loss on one sub-flow, recovery packets are immediately sent over another sub-flow that is not or less lossy *and* has spare congestion window slots. FUSO is *fast* in that it does not need to wait for timeout on the lossy sub-flow, and it is *cautious* in that it does not violate congestion control algorithm. Testbed experiments and simulations show that FUSO decreases the latency-sensitive flows' 99^{th} percentile FCT by up to ∼82.3% in a 1Gbps testbed, and up to ∼87.9% in a 10Gpbs large-scale simulated network.

1 Introduction

In recent years, large data centers have been built at an unforeseen rate and scale worldwide. Each data center may contain 100K servers, interconnected together by a large data center network (DCN) consisting of thousands of network equipments *e.g.*, switches and links. Modern applications hosted in DCN care much about the tail flow completion time (FCT) (*e.g.*, 99^{th} percentile). For example, in response to a user request, a web application (*e.g.*, Bing, Google, Facebook) often touches computation or memory resources of hundreds of machines, generating a large number of parallel latency-sensitive flows within the DCN. The overall application performance is commonly governed by the last completed flows [1, 2]. Therefore, the application performance will be greatly impaired if the network is *lossy*, as the tail FCT of TCP flows may greatly suffer from retransmission timeouts (RTO) [3, 4] under lossy condition.

Unluckily, packet losses are not uncommon even in well-engineered modern datacenter networks (§2.1). Conventionally, most of packet losses are due to buffer overflow caused by congestion, *e.g.*, incast [5, 6]. However, with the increasing deployment of the Explicit Congestion Notification (ECN) and fine-tuned TCP congestion control algorithm (*e.g.*, [1, 7]), the network congestion has been greatly mitigated (*e.g.*, from 1% to 0.01% [6]). But it still cannot be eliminated [7, 8]. Besides congestion, packets may also get lost due to failure (*e.g.*, malfunctioning hardware [3]). While normally hardware-induced loss rate is low (∼0.001%) [3], the rate can exceed 1% when hardware does not function properly. The reason for malfunctioning hardware is complex. It can come from ASIC deficits, or simply due to aging. Although the overall instances of malfunctioning hardware are small, once it happens, it usually takes hours or days to detect and mitigate [3].

We show, both analytically and experimentally, that even a moderate rise of loss rate (*e.g.*, to 1%) can already cause more than 1% of flows to hit RTOs (§2), and therefore greatly increases the 99^{th} percentile of flow FCT. Thus, we need a more robust transport that can ensure low tail FCT even when facing this adverse situation with lossy hardware. Previously, several techniques have been proposed to reduce TCP RTOs by adding more aggressiveness in loss recovery [4]. These schemes, originally designed for the Internet, have not been well tested in a DCN environment, where congestion may be highly correlated, *i.e.*, incast. Therefore, they are facing a difficult dilemma: if being too aggressive, this additional aggressiveness may offset the effect of the fine-tuned congestion control algorithm for DCN and induce congestion losses; Otherwise, being too timid would still cause delayed tail FCT.

In this paper, we advocate to utilize *multiple parallel paths*, which are plenty in most existing DCN topologies [6, 9–12], *to perform faster loss recovery*, without adding more congestion. To this end, we present *F*ast M*u*lti-path Lo*s*s Rec*o*very (FUSO), which employs multiple distinct paths for data transmission (similar to MPTCP [13–15]). FUSO fundamentally avoids the aforementioned dilemma of single-path TCP enhancements [4]. On one hand, FUSO strictly follows TCP congestion control algorithm which is well tuned for existing DCN. That is, a packet can leave the sender only when the TCP congestion window allows. Therefore, FUSO will behave equally aggressively as TCP flows (or precisely MPTCP flows). On the other hand, FUSO sender

*Dan Pei is the corresponding author.

will proactively (immediately) recover potential packet loss in a few paths (usually the "bad" paths) using other paths (usually the "good" paths). By exploiting the diversity of these paths, FUSO can keep the tail FCT low even with malfunctioning hardware. This behavior is fundamentally different from MPTCP, where each sub-flow is normally responsible to only recover its own losses. Although MPTCP provides an excellent performance for long flows' throughput, it may actually hurt the tail FCT of small flows compared to normal TCP (more discussion in §2.4).

Particularly, FUSO conducts proactive multi-path loss recovery as follows. When a sub-flow has no more new data to send, FUSO tries to utilize this sub-flow's spare resources permitted by transport congestion control to do proactive loss recovery on another sub-flow. FUSO speculates a path status from the information already recorded in the transport stack (*e.g.*, packet retransmission). Then it proactively transmits recovery packets through those good paths, to protect those packets suspected to be lost in the bad paths. By doing this, there is no need to wait for bad paths to recover loss by themselves which may cost a rather long time (*e.g.*, rely on timeout). Note that, because FUSO adds no aggressiveness to congestion control, even when loss happens at the edge (*e.g.*, incast) where no path diversity could be utilized, FUSO can still gracefully bound the redundancy incurred by proactive loss recovery, and offer a good performance (§5.2.3). The major contributions of the paper are summarized as follows.

1) We measure the attributes of packets loss in a Microsoft's production DCN. Then, through analysis and testbed experiments, we quantify the impact of packet loss on TCP FCT in DCN for the first time. We show that even a moderate rise of loss rate (*e.g.*, to 1%) would already cause enough flows (*e.g.*, >1%) to timeout to affect the 99^{th} percentile FCT.

2) We identify that the fundamental challenge for transport loss recovery in DCN is how to accelerate loss recovery under various loss conditions without causing congestion. We further show that existing loss recovery solutions differ just in their *fixed* choices of aggressiveness when dealing with the above challenge, and are not adaptive enough to deal with different loss conditions.

3) We design a novel loss transport recovery approach that exploits multi-path diversity in DCN. In our proposed solution FUSO, when loss is suspected on one sub-flow, recovery packets are immediately sent over another sub-flow that is speculated to be not or less lossy *and* has a spare congestion window. However, we show that, although conventional MPTCP [13–15] provides an excellent multi-path transport architecture and significantly improves the performance for long flows, it actually hurts the tail FCT for small flows.

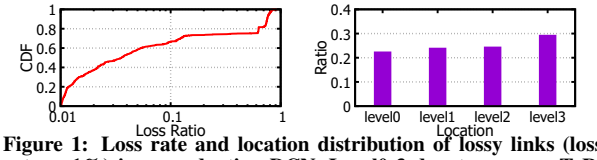


Figure 1: Loss rate and location distribution of lossy links (loss rate > 1%) in a production DCN. Level0-3 denote server↔ToR, ToR↔Agg, Agg↔Spine, and Spine↔Core, respectively.

4) We implement FUSO in Linux kernel with ∼900 lines of code (available at https://github.com/1989chenguo/FUSO). Experiment results show that FUSO's dynamic speculation-based loss recovery adapts to various loss conditions well. It decreases the latency-sensitive flows' 99^{th} percentile FCT by up to ∼82.3% in an 1Gbps testbed, and up to ∼87.9% in a 10Gpbs large-scale simulated network.

2 Fighting Against Packet Loss

2.1 Packet Loss in DCN

We first measure the attributes of packets loss in DCN, using *Netbouncer* within a Microsoft Azure's production data center. NetBouncer is a service deployed in Microsoft data centers for measuring link status. It is an end-host and switch joint solution and employs an active probing mechanism. End-hosts inject probing packets destined to network switches via IP-in-IP tunneling and switches bounce back the packets to the endhosts. It is an always-on service and the probing is done periodically. We have measured the packet loss in the data center for five days during December 1st-5th, 2015. The data center has four layers of switches, top-of-rack (ToR), Aggregation (Agg), Spine and Core from bottom to top.

Loss is not uncommon: In our operation experience, we find that although the portion of lossy links is small, they are not uncommon (also revealed in [3]). We define those links with loss rate (measured per hour) greater than 1% as *lossy links*, which may greatly impair the up-layer application performance (§2.2). Taking one day's data as an example, Fig. 1 (left part) shows the loss rate distribution among all lossy links during an hour (22:00-23:00). The mean loss rate of all the lossy links is ∼4%, and ∼63% of lossy links have the loss rate between 1% to 10%. About 22% of links even have a detected loss rate larger than 60%, where such exceptionally high loss rate maybe due to switch ASIC deficits (*e.g.*, packet black-hole [3]). We examine all the 5 days's data and find the loss rate distributions all very similar. It shows that although the portion of lossy link is small, they are the norm rather than the exception in large-scale data centers. Packet loss can be caused due to various reasons including failures and congestion.

Location of loss: Next, we analyze the location where packet loss happens. As shown in Fig. 1 (right part), among all the detected lossy links, there are only ∼22% of lossy links that are at the edge (server↔ToR, *i.e.*, level0), and ∼78% are happening in the network (above

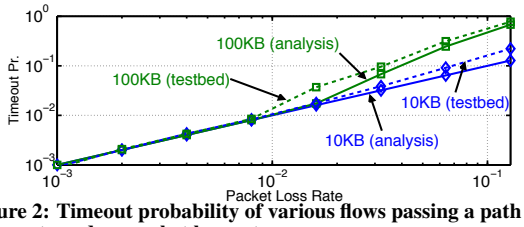


Figure 2: Timeout probability of various flows passing a path with different random packet loss rate.

ToR, *i.e.*, level1-3). About 22%, 24%, 25% and 29% of lossy links are located respectively at server↔ToR, ToR↔Agg, Agg↔Spine and Spine↔Core.

In summary, even in well-engineered modern data center networks, *packet losses are inevitable*. Although the overall loss rate is low, the packet loss rate in some areas (*e.g.*, links) can exceed several percents, when there are failures such as malfunctioning hardware or severe congestions. Moreover, *most losses happen in the network instead of the edge*.

2.2 Impact of Packet Loss

Once a packet gets lost in the network, TCP needs to recover it to provide reliable communication. There are two existing loss detection and recovery mechanisms in TCP [1]: *fast recovery* and *retransmission timeout (RTO)*. Fast recovery detects a packet loss by monitoring duplicated ACKs (or DACKs) and starts to retransmit an old packet once a certain number (*i.e.*, three) of DACKs have been received. If there are not enough DACKs, TCP has to rely on RTO and retransmits all un-ACKed packets after the timeout. To prevent premature timeouts and also limited by the kernel timer resolution, the RTO value is set rather conservatively, usually several times of the round-trip-time (RTT). Specifically, in a production data center, the minimum RTO is set to be $5ms$ [1,3] (the lowest value supported in current Linux kernel [16]), while the RTT is usually hundreds of μs [1, 3, 16]. As a consequence, for a latency-sensitive flow, which is usually small in size, encountering merely one RTO would already increase its completion time by several times and cause unacceptable performance degradation.

Therefore, the core issue in achieving low FCT for small latency-sensitive flows when facing packet losses is to avoid RTO. However, current TCP still has to rely on RTO to recover from packet loss in the following three cases [4, 17, 18]. i) The last packet or a series of consecutive packets at the tail of a TCP flow are lost (*i.e.*, *tail loss*), where the TCP sender cannot get enough DACKs to trigger fast recovery and will incur an RTO. ii) A whole window worth of packets are lost (*i.e.*, *whole window loss*). iii) A retransmitted packet also gets lost (*i.e.*, *retransmission loss*).

[1]Many production data centers also use DCTCP [1] as their network transport protocol. DCTCP has the same loss recovery scheme as TCP. Thus, for ease of presentation, we use TCP to stand for both TCP and DCTCP while discussing the loss recovery.

To understand how likely RTO may occur to a flow, we take both a simple mathematical analysis (estimated lower bound) and testbed experiments to analyze the timeout probability of a TCP flow with different flow sizes and different loss rates. We consider one-way random loss condition here for simplicity, but the impact on TCP performance and our FUSO scheme are by no means limited to this loss pattern (see §5).

Let's first assume the network path has a loss probability of p. Assuming the TCP sender needs k DACKs to trigger fast recovery, any of the last k packets getting lost will lead to an RTO. This tail loss probability is $p_{tail} = 1-(1-p)^k$. For standard TCP, $k = 3$, but recent Linux kernel which implement's *early retransmit* [19] reduces k to 1 at the end of the transaction. Therefore, if we consider early retransmit, the tail loss probability is simply p. The whole window loss probability can easily be derived as $p_{win} = p^w$, where w is the TCP window size. For retransmission loss, clearly, the probability that both the original packet and its retransmission are lost is p^2. Let x be the number of packets in a TCP flow. The probability that the flow encounters at least one retransmission loss is $p_{retx} = 1-(1-p^2)^x$. In summary, the timeout probability of the flow should be $p_{RTO} \geq max(p_{tail}, p_{win}, p_{retx})$. The solid lines in Fig. 2 show the analyzed lower bound timeout probability of a TCP flow with different flow sizes under various loss rates. Here, we consider the early retransmit ($k = 1$).

To verify our analysis, we also conduct a testbed experiment to generate TCP flows between two servers. All flows pass through a path with one-way random loss. *Netem* [20, 21] is used to generate different loss rate on the path. More details about the testbed settings can be found in §4.2 and §5. The dotted lines in Fig. 2 shows the testbed results, which verify that our analysis serves as a good lower bound of the timeout probability.

There are a few observations. Firstly, for tiny flows (*e.g.*, 10KB), the timeout probability linearly grows with the random loss rate. This is because the tail loss probability dominates. However, a tiny loss probability would affect the tail of FCT. For example, a moderate rise of the probability to 1% would cause a timeout probability larger than 1%, which means the 99^{th} percentile of FCT would be greatly impacted. Secondly, when the flow size increases, *e.g.*, ≥100KB, the retransmission loss may dominate, especially when the random hardware loss rate is larger than 1%. We can see a clear rise in timeout probabilities for the flows with 100KB in Fig. 2. In summary, we conclude that *a small random loss rate (i.e., >1%) would already cause enough flows to timeout to affect the 99^{th} percentile of FCT*. This can also explain why a malfunctioning switch in the Azure datacenter that drops ∼2% of the packets causes great performance degradation of all the services that traverse this switch [3].

2.3 Challenge for TCP Loss Recovery

To prevent timeout, when there are not enough returned DACKs to trigger fast recovery, prior work (*e.g.*, [4]) adds aggressiveness to congestion control to do loss recovery before RTO. However, deciding the *aggressiveness level*, *i.e.*, how long to wait before sending recovery packets, to adapt to complex network conditions in DCNs is a daunting task.

As introduced before, congestion and failure loss coexist in DCN. Congestion losses are very bursty and often lead to multiple consecutive packet losses [1, 4, 5, 7]. For congestion loss, recovery should be delayed for enough time before being sent out after the original packets. If a recovery packet is sent too fast before congestion disappears, the recovery packet may get dropped by the overflowed buffer and also worsen the congestion. However, for some failure loss such as random drop, recovery packets should be sent as fast as possible to accelerate the recovery process. Otherwise the delay for sending recovery packets already increases the FCT of latency-sensitive flows. Facing this difficult dilemma, previous schemes choose different *aggressiveness levels* in an ad-hoc manner, from a conservative 2RTT in Tail Loss Probe (TLP) [22], modestly conservative 1/4 RTT in TCP Instant Recovery (TCP-IR) [23], to a very aggressive zero time in Proactive [4]. Unfortunately, the fixed settings of aggressiveness levels make above existing schemes incapable of adapting to complex loss conditions: different loss characteristics under either congestion loss, failure loss or both.

Essentially, we identify that the fundamental challenge for transport loss recovery in DCN is *how to accelerate loss recovery as soon as possible, under various loss conditions without causing congestion*. Single-path loss recovery is not a promising direction to address this challenge because the recovery packets have to be sent over the same path that is under various loss conditions, the exact nature (congestion-induced, failure-induced, or both) of which are often unclear to the sender. One might think that through explicitly identifying congestion loss using schemes such as CP [24], transport can distinguish congestion and failure loss with the help of switches. However, there lacks a study on such design and its reliability under hardware failure conditions still remains to be an open question in complex production DCNs.

2.4 Utilizing Multi-path

Then it is natural to raise a question: why not try another good path when loss is speculated on one "bad" path? Actually, current DCN environment offers us a good chance to design a better loss recovery scheme based on multi-path. Current DCN provides many parallel paths (*e.g.*, 64 or more) between any two nodes by dense interconnected topologies [6, 9–12]. Usually, these paths have a big loss diversity due to different congestion and failure conditions. When a few paths are experiencing failure such as random loss or black-hole, the rest paths (*i.e.*, the majority) may remain in a good state without failure loss. Also, caused by uneven load balance [25], some paths may be heavily congested to drop packets while other paths are in light load.

One might think that using multi-path transport protocol such as MPTCP [13–15] is able to address the challenge above. On the contrary, although MPTCP provides excellent performance for long flows, it actually hurts the tail FCT of small latency-sensitive flows under lossy condition (see §5). It is because that, while MPTCP explores multiple paths, each of its paths normally has to recover loss by itself. Therefore, its overall completion time depends on the last completed sub-flow on the worst path. Simply exploring multiple paths actually increases the chance to hit the bad paths. Therefore, MPTCP's lack of an effective loss recovery mechanism leads to a long tail FCT especially for small flows.

To this end, we propose *F*ast M*u*lti-path Lo*s*s Rec*o*very (FUSO), which leverages multi-path diversity for transport loss recovery. FUSO fundamentally avoids the aforementioned dilemma (§2.3), by utilizing those paths in good status to proactively (or immediately) conduct loss recovery for bad paths. First, FUSO is *cautious* in that it strictly follows TCP congestion control algorithm that is tuned for existing DCN, adding no aggressiveness. Second, FUSO is *fast* in that the sender will proactively recover potential packet loss in bad paths using good paths before timeout. As shown before, most losses happen in the network (§2.1), which gives plenty of opportunities for FUSO to leverage multi-path diversity. On the other hand, sometimes packet losses may happen at the edge (*e.g.*, incast) due to congestion, where there is no path diversity that can be utilized for multi-path loss recovery. Thanks to strictly following the congestion control, FUSO can adaptively throttle its proactive loss recovery behaviour and be conservative to avoid worsening the congestion (see §5.2.3).

Note that there is a mechanism named opportunistic retransmission [14] in MPTCP, which may also trigger proactive retransmission through alternative good subflows similar to the scheme in our FUSO solution. Although sharing the same high-level idea which is utilizing path diversity, it addresses different problems from FUSO. MPTCP opportunistic retransmission is designed for wide-area network (WAN) to maintain a high throughput and minimize the memory (receive or send buffer) usage, to cope with severe reordering caused by diverse delay of multiple paths. It is triggered only when the new data cannot be sent because the receive window or the send buffer is full. It immediately retransmits the oldest un-ACKed packet through alternative good paths which have the smallest RTT. Although opportunistic re-

Algorithm 1 Proactive multi-path loss recovery.

```
1: function TRY_SEND_RECOVERIES( )
2:     while BytesInFlight_Total < CWND_Total and no new data do
3:         return ← SEND_A_RECOVERY( )
4:         if return == NOT_SEND then
5:             break
1: function SEND_A_RECOVERY( )
2:     FIND_WORST_SUB-FLOW( )
3:     FIND_BEST_SUB-FLOW( )
4:     if no worst found or no best sub-flow found then
5:         return NOT_SEND
6:     recovery_packet←one un-ACKed packet of the worst sub-flow
7:     Send the recovery_packet through the best sub-flow
8:     BytesInFlight_Total += Size_recovery_packet
```

transmission helps to achieve a high throughput for long flows in WAN scenario, it offers little help on maintaining a low FCT under lossy condition in DCN scenario where paths often have very similar delay. More importantly, in DCN those latency-sensitive flows are often with too small sizes (*e.g.*, <100KB) to cause severe reordering, which cannot eat up the end-host's buffer. Therefore, these small flows cannot trigger the opportunistic retransmission.

3 FUSO Design

3.1 Overview

We now introduce FUSO. FUSO is built on top of the multi-path transport, in which a TCP flow is divided into multiple sub-flows. Note that FUSO focuses on multi-path loss recovery rather than multi-path congestion control. Particularly, in this paper, we build FUSO on MPTCP[2] [13–15]. ECMP [26] or SDN methods (*e.g.*, XPath [27]) can be used to implicitly or explicitly map the sub-flows[3] onto different physical paths in DCN.

The core scheme of FUSO is that, by strictly following the congestion control, if there is a spare congestion window (*cwnd*), FUSO first tries to transmit new data. If the up-layer application currently has no new data, FUSO utilizes this transmission opportunity to proactively/immediately transmit recovery packets for those suspected lost (un-ACKed[4]) packets on "bad" sub-flows, by utilizing "good" sub-flows. Note that FUSO does not affect the existing MPTCP opportunistic retransmission mechanism triggered by full receive window. These two mechanisms can be complementary to each other.

We separately discuss the FUSO sender and receiver for better clarification. In a FUSO connection, the sender and receiver refer to the end hosts sending data and the ACK respectively. Both ends are simultaneously the sender and receiver in a two-way connection.

[2]FUSO can also work on other multi-path transport protocols.

[3]We use 'sub-flow' and 'path' interchangeably in this Section.

[4]For TCP with SACK [28] enabled, un-ACKed packets refer to those un-SACKed and un-ACKed ones.

3.2 FUSO Sender

The FUSO sender's proactive multi-path loss recovery process can be summarized as Algo. 1. Specifically, we insert a function *TRY_SEND_RECOVERIES()* in the transport stack, monitoring the changes of $BytesInFlight_{Total}$, $CWND_{Total}$ and the application data. This function needs to be inserted into two positions: i) after all the data delivered from the application has been pushed into the transport send buffer and sent out, which indicates that there is currently no more new data delivered from the up-layer application; ii) after an ACK is received and the transport status (*e.g.*, $BytesInFlight_{Total}$, $CWND_{Total}$) has been changed. More implementation-related details are discussed in §4.1. Within this function, the sender calls the function *SEND_A_RECOVERY()* to send a recovery packet if the following conditions are both satisfied: i) there is spare window capacity allowed by congestion control, *and* ii) all new data has been sent out.

In the function *SEND_A_RECOVERY()*, FUSO sender first calls the function *FIND_WORST_SUB-FLOW()* and *FIND_BEST_SUB-FLOW()* to find the current worst and best sub-flows. The worst sub-flow is selected only among those who have *un-ACKed data*, and the best sub-flow is selected only among those whose *congestion window (cwnd) has spare spaces* permitted by congestion control. We defer the discussion on how to find the worst and best paths to §3.2.1.

If currently there is no worst or no best sub-flow, FUSO stops generating recovery packets for this round. Next, if the worst *and* best sub-flows are found, a recovery packet for the worst sub-flow is generated. Because FUSO conducts proactive loss recovery before a packet is detected as lost either by DACKs or RTO, we have to *guess* which packet is most likely to be the lost one. FUSO infers the packet as the oldest un-ACKed packet which has been sent out for the longest time. Thus, the sender proactively generates a recovery packet for one un-ACKed packet on the worst path in the ascending order of TCP sequence number (*i.e.*, the oldest packet in this path). To avoid adding too much unnecessary traffic to the network, an un-ACKed packet will be sent at most once by the proactive loss recovery scheme in FUSO.

After the recovery packet is generated, FUSO sender sends it through the best sub-flow. Note that the recovery packet is regarded as a *new data packet* for the best sub-flow. The recovery packet is under the best sub-flow's congestion control, and, if it gets lost in the best sub-flow, it will be retransmitted as normal packets in the best sub-flow using the standard TCP loss recovery. However, to avoid duplicate recovery, these packets will not be counted in the un-ACKed packets waiting for recovery when FUSO sender conducts fast multi-path loss recovery later. In the last step of *SEND_A_RECOVERY()*, $BytesInFlight_{Total}$ is incremented and the conditions

in the while loop in *TRY_SEND_RECOVERIES()* will be checked again.

3.2.1 Path Selection

Whenever congestion control offers a chance to transmit packets, FUSO tries to proactively recover the suspected lost packet in the currently "worst" path which is most likely to encounter packet loss, utilizing the currently "best" path which is least likely to encounter packet loss. Therefore, we define a metric $C_l = \alpha \cdot \overline{lossrate} + \beta \cdot lossrate_{last}$, to describe the possibility of packet loss happening in a sub-flow. C_l is the weighted sum of the overall packet loss rate $\overline{lossrate}$ and the most recent packet loss rate $lossrate_{last}$ in this sub-flow. α and β are the respective weight of each part. Since current TCP/MPTCP retransmits a packet after detecting it as lost either by DACK or RTO, FUSO uses the ratio of total retransmitted packets *to* the total transmitted packets as the approximation of $\overline{lossrate}$. Note that recovery packets generated by FUSO are regarded as new packets instead of retransmitted packets for sub-flows. $lossrate_{last}$ is calculated as the ratio of one *to* the number of transmitted packets from (including) the last retransmission.

The worst sub-flow is picked among those which have at least one un-ACKed packet (possibly lost), and with the *largest* C_l. For sub-flows which have never encountered a retransmission yet, their C_l equals zero. If all sub-flows' C_l equals zero, FUSO picks the one with the largest measured RTT thus to optimize the overall FCT.

The best sub-flow is picked among those which have spare *cwnd*, and with the *smallest* C_l. For sub-flows never encountering a retransmission yet, their C_l equals zero and is smaller than others. If more than one sub-flows have zero C_l, FUSO picks the one with the smallest measured RTT as the best sub-flow. Note that at the initial state, some sub-flows may have never transmitted any data when FUSO starts proactive loss recovery. Then these sub-flows' C_l equal infinity and have the least priority when FUSO selects the best sub-flow. If all sub-flows' C_l equal infinity, FUSO randomly picks one as the best sub-flow. Note that the best and worst sub-flows may be the same one. Under this condition, FUSO simply transmits the recovery packets in the same sub-flow after the original packets.

3.3 FUSO Receiver

FUSO receiver is relatively simple. In multi-path transport protocol such as MPTCP, the receiver has a data-level (*i.e.*, flow-level) receive buffer and each sub-flow has a virtual receive buffer that is mapped to the data-level receive buffer. Upon receiving a FUSO recovery packet, FUSO receiver directly inserts the recovery packet into the corresponding position of the data-level receive buffer, to complete the flow transmission. The FUSO recovery packets will not affect the bad sub-flows' behaviour on the sub-flow-level, but directly recovery the lost packets on the data-level.

For the best sub-flow that transmits FUSO recovery packets, these packets are naturally regarded as normal data packets in terms of this sub-flow's congestion control and original TCP loss recovery. Although protected by them, the bad sub-flow is not aware of these recovery packets, and may unnecessarily retransmit the old data packet (if lost) itself. FUSO currently chooses such a simple approach to maintain the exact congestion control behavior and add no aggressiveness, both on individual sub-flows and the overall multi-path transport flow. It needs no further coordination besides the original ACK schemes in TCP between the sender and the receiver, but may incur some redundant retransmissions. A naive solution to eliminate the redundant retransmissions may be that the receiver proactively generates ACKs for the original packets in the bad sub-flow, upon receiving recovery packets from other good sub-flows. However, this may cause adverse interaction with congestion control. Specifically, the bad sub-flow's sender end may wrongly judge the path as in a good status and increases its sending rate, which may exacerbate the loss.

In order to maintain the congestion control behavior *and* eliminate the redundant retransmissions, it may need very complex changes to the MCTCP/TCP protocols. The sender and receiver must coordinate to decide whether/how it should change each sub-flow's congestion control behavior (*e.g.*, increase/decrease how much to the *cwnd*), to cope with various conditions, such as i) the proactive retransmission received but the original packet lost, ii) the original packet received but the proactive retransmission lost, iii) both packets lost, iv) the proactive retransmission received before the original packet, v) the original packet received before the proactive retransmission, *etc.* The feasibility of such a solution and how to design it still requires further study and is left as our future work. FUSO currently chooses to trade a little redundant retransmission (see §5.2.2 and 5.2.3) for the aforementioned simple and low-cost approach.

4 Implementation and Testbed Setup

4.1 FUSO Implementation

We implemented FUSO in Linux kernel 3.18 with 827 lines of code, building FUSO upon MPTCP's latest Linux implementation (v0.90) [29].

FUSO sender: We insert *TRY_SEND_RECOVERIES()* in Algo. 1 into the following positions of the sender's transport kernel, to check whether a FUSO recovery packet should be sent now: 1) in function *tcp_sendmsg()* after all the data delivered from the application has been pushed into the send buffer; 2) in function *tcp_v4_rcv()* after an ACK is received and the transport status (*cwnd*, bytes in flight *etc.*) has been changed.

In *TRY_SEND_RECOVERIES()*, FUSO detects that there is currently no more new data from the up-layer application, if the two conditions are both satisfied: i) the data delivered from the application has all been pushed in the send buffer; ii) the packets in the send buffer have all been sent. If a multi-path loss recovery packet is allowed to be sent, FUSO sender calls the function *SEND_A_RECOVERY()* and picks one un-ACKed packets (in ascending order of sequence number) on the worst sub-flow, then copies and transmits it on the best sub-flow. We utilize existing functions in MPTCP to reconstruct the mapping of the recovery packet's data-level (*i.e.*, flow-level) sequence number to the new sub-flow-level sequence number. Also, FUSO sender remembers this packet to ensure that each un-ACKed packet is protected for at most once. In FUSO, both the formats of data packets and FUSO recovery packets have no difference from those in the original MPTCP protocol. The data-level sequence number (DSN) in the existing Data Sequence Signal (DSS) option of MPTCP header can notify the receiver how to map this recovery packet into data-level data.

It is noteworthy that, besides the opportunistic retransmission introduced before, original MPTCP may also retransmit the data packets originally delivered to one sub-flow through other sub-flows under the following condition: if one sub-flow is judged to be dead when it encounters certain number of consecutive timeouts, all the packets once distributed to this sub-flow will be re-injected to a special flow-level sending buffer called "reinject queue". Then MPTCP will redistribute these packets to other sub-flows. This is a failover scheme to deal with the case that some of its sub-flows completely fail. However, it is too slow (after a sub-flow is dead) to provide a low FCT under lossy conditions.

FUSO receiver: The receiving process has already been implemented in MPTCP's original receiving logic, which requires no other modification. According to the DSN in the header option, the receiver will insert the multi-path loss recovery packet in the corresponding position of the data-level receive buffer, and complete the data transmission. Note that in the current MPTCP's Linux implementation, the receiver only hands over packets to the data-level receive buffer which are in-sequence in the sub-flow level, and buffers the packets which are out-of-sequence (OoS) in the sub-flow level in the sub-flow's OoS queue. This implementation reduces the reordering computation overhead, but may severely defer the completion time of the overall MPTCP flow. Since packets may be retransmitted by other sub-flows, those packets OoS in sub-flow level may be in-sequence in the data level. As such, in-sequence data-level packets may not be inserted to the data-level receive buffer even when they arrive at the receiver, because of being deferred by the former lost packets in the same sub-flow. To solve this problem, we implement a minor modification to current MPTCP's receiving end implementation, which immediately copies the sub-flow-level OoS packets directly to the MPTCP data-level receive buffer. This receiving end modification is 34 lines of code.

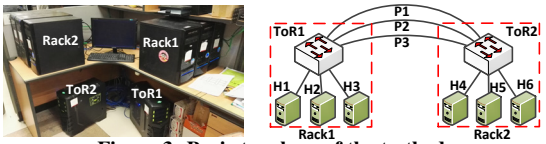


Figure 3: Basic topology of the testbed.

4.2 Testbed Setup

We build a small 1Gbps testbed as shown in Fig. 3. It consists of two 6-port ToR switches (*ToR1, ToR2*) and six hosts (*H1* $\sim$*H6*) located in the two racks below the ToR switches. There are three parallel paths between the two racks, emulating the multi-path DCN environment.

Each host is a desktop with an Intel E7300 Core2 Duo 2.66GHz CPU, 4GB RAM and 1Gbps NIC, and runs Ubuntu 14.04 64-bit with Linux 3.18.20 kernel. We use two servers to emulate the two ToR switches. Each server-emulated switch is a desktop with an Intel Core i7-4790 3.60GHz CPU, 32GB RAM, and 7 Intel I350 Gigabit Ethernet NICs (one reserved for the management). All server-emulated switches run Ubuntu 14.04 64-bit with Linux 4.4-RC7 kernel. Originally, current Linux kernel only support IP-address-based (<*src,dst*> pair) ECMP [26] when forwarding packets. Therefore, we made a minor modification (8 lines of code) to the switches kernel, thus to enable layer-4 port-based ECMP [26] (<*src,dst,sport,dport,protocol*> pair) which is widely supported by commodity switches and used in production DCNs [3, 6, 16].

Flows are randomly hashed to the physical paths by ECMP in our testbed. Each switch port buffer size is 128KB. The basic RTT in our testbed is $\sim$280μs. ECN is enabled using Linux *qdisc RED* module, with marking threshold set to be 32KB according to the guidance by [7]. We set TCP minRTO to 5*ms* [1, 3]. These settings are used in all the testbed experiments .

5 Evaluation

In this section, we use both testbed experiments and ns-2 simulations to show the following key points. 1) Our testbed experiments show ***FUSO's good performance under various lossy conditions, including failure loss, failure & congestion loss, and congestion loss.*** 2) We also use targeted testbed experiments to ***analyze the impact of sub-flow number on FUSO's performance.*** 3) Our detailed packet-level simulations confirm that ***FUSO scales to large topologies.***

5.1 Schemes Compared

We compare the following schemes with FUSO in our testbed and simulation experiments. For the simulations,

we implement all the following schemes in ns-2 [30] simulator. For the testbed, we implement Proactive and Repflow [31] in Linux, and directly use the source codes of other schemes.

TCP: The standard TCP acting as the baseline. We enable the latest loss recovery schemes in IETF RFCs for TCP, including SACK [28], SACK based recovery [18], Limited Transmit [32] and Early Retransmission [19]. The rest of the compared schemes are all built on this baseline TCP.

Tail Loss Probe (TLP) [22]: The latest single-path TCP enhancement scheme using prober to accelerate loss recovery. TLP transmits one more packet after 2 RTTs when no ACK is received at the end of the transaction or when the congestion window is full. This extra packet is a prober to trigger the duplicate ACKs from the receiver before timeout.

TCP Instant Recovery (TCP-IR)[5] [23]: The latest single-path TCP enhancement scheme using both prober and redundancy. It generates a coded packet for every group of packets sent in a time bin, and waits for 1/4 RTT to send it out. This coded packet protects a single packet loss in this group providing "instant recovery", and also acts like a prober as in TLP. According to the authors' recommendation [4], we set the coding timebin to be 1/2 RTT and the maximum coding block to be 16.

Proactive [4]: A single-path TCP enhancement scheme to accelerate loss recovery by duplicating every TCP data packet. We have implemented Proactive in Linux kernel 3.18.

MPTCP [15]: The state-of-the-art multi-path transport protocol. We use the latest Linux version of MPTCP implementation (v0.90 [29]), which includes the opportunistic retransmission mechanism [14].

RepFlow [31]: A simple multi-path latency improvement scheme by proactively transmitting two duplicated flows. We have implemented RepFlow in the application layer according to [31].

For all compared schemes, the initial TCP window is set to 16 [3]. Note that for FUSO and MPTCP, the initial window of each sub-flow is set to be $\frac{16}{number\ of\ subflows}$, which forms the same 16 initial window in total for a connection. Unless specified otherwise, we configure 4 sub-flows for each FUSO and MPTCP connection in the testbed experiments, which offers the best performance for both methods in various conditions. We compare the performance of FUSO/ MPTCP using different number of sub-flows in §5.2.4. FUSO's path selection parameters α, β (§3.2.1) are both set to be 0.5.

[5]TCP-IR has published its code [33] and we successfully compiled it to our testbed hosts. However, after trying various settings, we are not able to get it running on our testbed environment due to some unknown reasons. As such, we evaluate TCP-IR only in simulation experiments.

5.2 Testbed Experiments

Benchmark Traffic: Based on the code in [34], we develop a simple client-server application. Each client sends requests to some randomly chosen servers for a certain size of data, with inter arrival time obeying the Poisson process. There are two types of requests from the client, 1) latency-sensitive queries with data sizes smaller than 100KB, and 2) background requests with sizes larger than 100KB. All the requests' sizes are sampled from two real data center workloads, web-search [1] and data-mining [10]. Each client initiates 10 long-lived transport connections (5 for latency-sensitive queries, and 5 for background requests) to each server, and round-robinly distributes the requests on each connection (of their type) to the server. We generate different loads through adjusting the requests' inter arrival time. All 6 hosts run both client and server processes. We separately enable the various compared schemes to serve the connections for latency-sensitive queries, and use standard TCP for the rest of connections for background requests. Note that before all evaluations, we generate 100KB data to warmup each FUSO/MPTCP connection and wait for an idle time to reset the initial window, thus to activate all the sub-flows. We compare the request completion time[6] of those latency-sensitive queries.

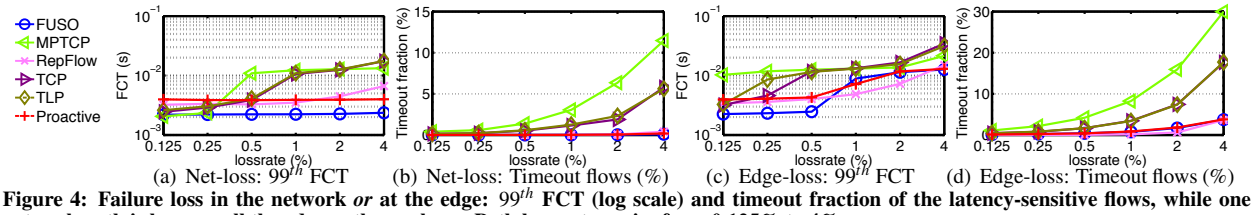
Emulating failure loss: We use *netem* [20,21] to generate failure packet loss with different loss patterns and loss rates. The network and edge loss are emulated by enabling *netem* loss module on certain network interfaces (on the switches or hosts). Two widely-used loss patterns are evaluated, random loss and bursty loss [35].

Due to space limitation, we only present the testbed results under random loss using web-search workload. We have evaluated FUSO under various settings, with different loss models (random and bursty [35]) using different workloads (web-search and data-mining), and FUSO consistently outperforms other schemes (reduce the latency-sensitive flows' 99^{th} percentile FCT by up to ~86.2% under bursty loss and data-mining traffic). All the experiment results are from 10 runs in total, with 15K flows generated in each run.

5.2.1 Failure Loss

We first show how FUSO can gracefully handle failure loss in DCN. To avoid the interference of congestion, no background traffic is injected, and we deploy the clients on $H4$-$H6$ generating small latency-sensitive requests (data size $<$ 100KB) respectively to $H1$-$H3$ without edge contention. We only focus on the failure loss in this experiment, and later we will show how FUSO performs when failure and congestion coexist. The requests are generated in an average load of 10Mbps [1].

[6]We use 'flow' and 'request', 'request completion time' and 'FCT' interchangeably in §5.

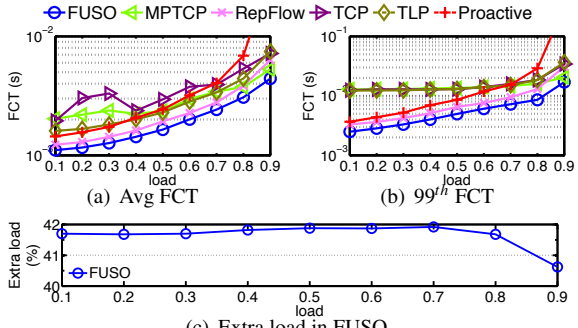


(a) Net-loss: 99^{th} FCT (b) Net-loss: Timeout flows (%) (c) Edge-loss: 99^{th} FCT (d) Edge-loss: Timeout flows (%)

Figure 4: Failure loss in the network *or* at the edge: 99^{th} FCT (log scale) and timeout fraction of the latency-sensitive flows, while one network path is lossy *or* all the edge paths are lossy. Path loss rate varies from 0.125% to 4%.

Loss in the network: We first evaluate the condition when failure loss happens in the network, by deliberately generating different loss rate for the path $P1$ in Fig. 3. Note that the two directions of $P1$ both have the same loss rate. Fig. 4(a) and Fig. 4(b) present the 99^{th} percentile FCT and the fraction of timeout ones among all the latency-sensitive flows. The results show that FUSO maintains both very low 99^{th} percentile FCT (<2.4ms) and fraction of timeout flows (<0.096%), as the path loss rate varies from 0.125% to 4%. FUSO reduces the 99^{th} percentile FCT by up to ∼82.3%, and the timeout fraction up to 100% (no timeout occurs in FUSO), compared to other schemes. The improvement is due to the multi-path loss recovery mechanisms of FUSO, which can explore and utilize good paths that are not lossy, and also makes the FCT depend on the best path explored. Although MPTCP also explores multiple paths, each of its paths normally has to recover loss by itself (more details in §2.4). Therefore, its overall completion time depends on the last completed sub-flow on the worst path. Lacking an effective loss recovery mechanism actually lengthens the tail FCT in MPTCP, as exploring multiple paths actually increases the chance to hit the bad paths. RepFlow offers a relatively better performance than other schemes by excessively duplicating every flow. However, this way of redundancy is actually less effective than FUSO, because each flow independently transmits data with no cooperative loss recovery as in FUSO. Since there is still a big chance for ECMP to hash the two duplicated flows into the same lossy path, it makes RepFlow have an ∼32%-64.5% higher 99^{th} percentile FCT than FUSO. Proactive also behaves inferiorly, suffering from similar problems as RepFlow. TLP performs almost the same as TCP because it sacrifices the ability to prevent timeouts in order to keep low aggressiveness.

Loss at the edge: We then evaluate the extreme condition when severe failure loss happens at the edge, by deliberately generating different loss rates for all the access links of $H4$-$H6$. We try to investigate how FUSO performs when sub-flows cannot leverage diversity among different physical paths. Fig. 4(c) and Fig. 4(d) show the results. Even with all sub-flows passing the same lossy path, FUSO still can maintain a consistent low timeout fraction to be under 0.8% when the loss rate is below 1%. However, the timeout fraction of other approaches except RepFlow and Proactive exceeds 3.3% at the same loss



(a) Avg FCT (b) 99^{th} FCT

(c) Extra load in FUSO

Figure 5: Failure & congestion loss: Average and 99^{th} FCT (log scale) of latency-sensitive flows, and the average extra load of all FUSO flows. Each flow's extra load is calculated by *the extra bytes incurred by FUSO* divided by *the total bytes transmitted*.

rate. As such, FUSO reduces the 99^{th} percentile FCT by up to ∼80.4% compared to TCP, TLP and MPTCP. When loss rate exceeds 2%, FUSO still maintains the 99^{th} FCT under 12.7ms. Although all sub-flows traverse the same lossy path in this scenario, the chance that all four of them simultaneously hit the loss has been decreased. FUSO can leverage the sub-flow which does not encounter loss currently to help recover lost packets in the sub-flow which hits loss at this moment. Due to the excessive redundancy, RepFlow and Proactive perform the best in this scenario when loss rate is high, but hurt the 99^{th} FCT when the loss rate is low. Later (§5.2.3) we will show that this excessive load and the non-adaptive redundancy ratio will substantially degrade the performance of latency-sensitive flows, when the congestion is caused by themselves such as in the incast [5] scenario.

5.2.2 Failure & Congestion Loss

Next we evaluate that how FUSO performs with co-existing failure and congestion loss. We generate a 2% random loss rate on both directions of path $P1$, which is similar to a real Spine switch failure case in production DCN [3]. We deploy the aforementioned clients on the $H4$-$H6$ and configure them to generate small latency-sensitive queries as well as background requests, to the servers randomly chosen from $H1$-$H3$. This cross-rack traffic [13, 36] ensures that all the flows have a chance going through the lossy network path. We inject different traffic load from light (0.1) to high (0.9), to investigate how FUSO performs from failure-loss-dominated scenario to congestion-loss-dominated scenario.

Results: Fig. 5 shows the results. Under conditions where failure and congestion loss coexist, FUSO main-

tains both very low average and 99th percentile FCT of latency-sensitive flows, from light to high load. Compared to MPTCP, TCP and TLP, FUSO reduces the average FCT by ∼28.2%-47.1%, and the 99th percentile by ∼17.2%-80.6%. FUSO also outperforms RepFlow by ∼10%-30.3% in average and ∼20.1%-44.8% in tail, due to two reasons: 1) the chance of two replicated flows in RepFlow being hashed to the same lossy path is non-negligible, and 2) excessive redundancy RepFlow adds congestion when load is high. As for Proactive, the replicated packets always go through the same path as the original data packets, which makes them share the same loss rate and further decrease its redundancy efficiency compared to RepFlow. Moreover, the simple duplicating behaviour extremely degrades its performance under heavy load. On the contrary, FUSO's proactive multi-path loss recovery helps to recover the congestion and failure loss *fast*, meanwhile remaining *cautious* to avoid adding aggressiveness. Even at the high load of 0.9, FUSO maintains the average and tail FCT to be below 4.5ms and 17.1ms, respectively. TCP behaves inferiorly due to coexisting severe congestion and failure loss, while MPTCP performs better in this case. TLP's faster loss recovery by adding moderate aggressiveness makes it perform better than both TCP and MPTCP.

We show the average extra load of all FUSO flows in Fig. 5(c). Each flow's extra load is calculated by *the extra bytes incurred by FUSO* divided by *the total bytes transmitted*. The results show that FUSO's fast loss recovery behaviour can gracefully adapt to the network condition. Particularly, when the load is low, FUSO generates relatively more recovery packets (∼42% extra load) to proactively recover the potential loss. Such relative high redundancy rate does not affect the FUSO flows' FCT, because that FUSO only generates redundancy utilizing the opportunity when the network is not congested (detected from spare *cwnd*) and there is no more new data. As the congestion becomes severe, FUSO naturally throttles the redundancy generation (down to ∼40% in 0.9 load) by strictly following the congestion control behaviour. Later (§5.2.3) we will show that FUSO generates even lesser redundancy when the network is more congested. Note that although FUSO's redundancy helps to improve the latency flows' FCT, they do incur some extra load to the network and slightly affect the overall network throughput. Compared with MPTCP, which uses the same congestion control as FUSO but incurs zero extra load, the average throughput of all flows in FUSO is ∼7.8% lower to ∼3.4% higher at various loads.

5.2.3 Congestion Loss: Incast

Now, we focus on the congestion loss at the edge which is a very challenging scenario for FUSO, to investigate whether FUSO is cautious enough to avoid adding congestion when there is no spare capacity in the bottleneck link. We deploy a receiver application on $H1$ to simultaneously generate requests to a number of sender applications deployed on $H2$-$H6$. After receiving the request, each sender immediately responds with a 64 KB data using the maximum sending rate. This traffic pattern, which is called incast [5], is very common in MapReduce-like [37] applications. We use all physical sending hosts in our testbed to emulate multiple senders [38]. We measure the completion time when all the responses have been successfully received. In this case we do not inject failure loss.

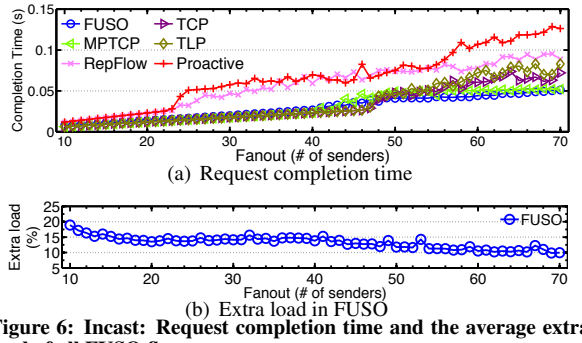
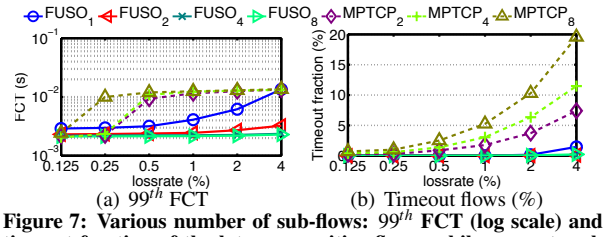


Figure 6: Incast: Request completion time and the average extra load of all FUSO flows.

Results: Fig. 6(a) shows the request completion time as the number of senders (*i.e.*, fanout) grows. When the fanout is below 44, FUSO, MPTCP, TCP and TLP behave similarly. As studied before [24, 39], a small min-RTO and appropriate ECN setting can offer a fairly good performance for standard TCP in the incast scenario. Because the total response size equals 64KB×fanout, the completion time linearly grows as the fanout increases. When fanout grows above 44, timeout occurs in MPTCP, which leads to a sudden rise of completion time. It is due to the relatively high burstiness caused by multiple sub-flows. However, FUSO's multi-path loss recovery scheme compensates this burstiness and remains an approximately linear growth of completion time in FUSO. The performance begins to degrade for all methods when the fanout exceeds 48. FUSO keeps performing the best, and keeps the completion time below 51.2ms even when the fanout becomes 70.

RepFlow and Proactive always take roughly twice the time to complete the query even when fanout is low (*e.g.*, <23), because they duplicate every flow (or packet) and add a certain excessive extra load to the network. As the fanout becomes larger, many timeouts occur and significantly impair the performance of them. For example, for a fanout of 30, RepFlow and Proactive need ∼47ms and ∼58ms to complete the request, respectively, while FUSO only needs less than 25ms. Although duplicating small flows can help to improve their performance under some lossy cases, it is not adaptive to complicated DCN environments, and even deteriorates the performance especially when the network is congested by



(a) 99^{th} FCT (b) Timeout flows (%)

Figure 7: Various number of sub-flows: 99^{th} FCT (log scale) and timeout fraction of the latency-sensitive flows, while one network path is lossy.

the small flows themselves. On the contrary, Fig. 6(b) shows that FUSO can gracefully adapt to the network condition and throttle the extra load in such extremely congested scenarios.

5.2.4 Various Number of Sub-flows

Now, we investigate the impact of the number of sub-flows on FUSO's performance. The settings are the same as in the network loss experiment in §5.2.1. We compare FUSO with 1,2,4 and 8 sub-flows, denoted as $FUSO_{1,2,4,8}$. Note that $FUSO_1$ simply retransmits the suspected lost packets in the same flow after the original packets before standard TCP loss recovery begins, without using multi-path.

Results: Fig. 7 shows the results. We can see that FUSO behaves better as it explores more paths using more sub-flows. Only adding redundancy without leveraging multi-path diversity causes the inferior performance of $FUSO_1$. $FUSO_4$ can offer a fairly good performance that is very close to $FUSO_8$, which means 4 sub-flows is enough for our small testbed with 3 parallel paths. On the contrary, MPTCP behaves worse as the number of sub-flows grows, lagged by the last completed sub-flow on the worst path (see §2.4).

5.3 Large-scale Simulations

Simulation settings: Besides testbed experiments, we also use ns-2 [30] to build a larger 3-layer, 4-pod simulated Fat-tree topology. The topology consists of 4 Spine switches and 4 pods below them, each containing 2 Aggregation and 2 ToR switches. All switches have 4 40Gbps ports, and each ToR switch has a rack of 8 10Gbps servers below. Each switch has 1MB buffer per port, with ECMP and ECN enabled. The whole topology contains 20 switches and 64 servers, *i.e.*, the largest scale for detailed packet-level simulation that could be finished in an acceptable time on our servers. The base RTT without queueing is $\sim$240μs. Given that, the ECN threshold is set to be 300KB [7]. We set the TCP min-RTO to be 5*ms* [3, 16]. The input traffic is generated the same as in §5.2.2, letting all the clients request both latency-sensitive queries and background data from randomly chosen servers. Besides web-search [1], we also evaluate another empirical data-mining workload [10]. Both FUSO and MPTCP use 8 sub-flows to adapt to the large topology. The results are from 10 runs in total, with 32K flows generated in each run.

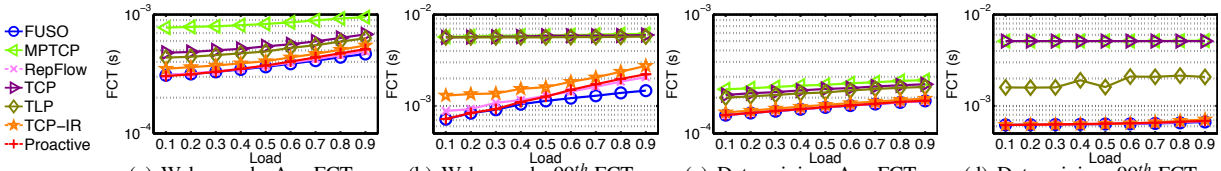
Empirical failure loss: To emulate the real condition in production data centers, we randomly set 5% links to be lossy. The loss rate of each lossy link is sampled from the distribution measured in §2.1 (Fig. 1(a)). Note that we have excluded the part in the distribution with exceptionally high loss rate (right most part in Fig. 1(a) with loss rate $>$ 60%) for sampling. It is because that standard TCP flows almost cannot finish and often upper-layer applications operations are triggered (*e.g.*, requesting resources from other machines) under such high loss rates. We randomly generate those lossy links at different locations including the edge and network, according to the real location distribution[7] in §2.1 (Fig. 1(b)).

Results: The results in Fig. 8 confirm that FUSO can gracefully scale to large topologies and complex lossy conditions. Under all loads, the average FCT of FUSO is $\sim$10.4%-60.3% lower than TCP, MPTCP, TLP and TCP-IR in web search workload, and $\sim$4.1%-39.4% lower in data mining workload. Also, the 99^{th} percentile is $\sim$29.2%-87.4% and $\sim$0%-87.9% lower in the two workloads respectively. TCP-IR chooses a more aggressive loss recovery manner than TLP. This improves the performance, but TCP-IR still has $\sim$29.2%-46.5% and $\sim$0%-6.1% higher 99^{th} FCT than FUSO under two workloads, respectively. Lacking multi-path makes TCP-IR's loss recovery less efficient, because the recovery packets may be also dropped while traversing the same lossy path as the former dropped data packets. Compared with RepFlow and Proactive which use certain excessive redundancy rate, FUSO still has up to $\sim$33.9% and $\sim$2.6% better 99^{th} percentile FCT under the two workloads respectively, due to the reasons discussed before. Because the simulated topology has a much higher capacity in the fabric link (40G) than the access link (10G), the congestion is significantly alleviated compared to the small testbed topology in §5.2.2. Thus TCP performs better than MPTCP for small flows in this scenario, because their performance depends more on the failure loss.

6 Discussion

FUSO follows the principle of prioritizing new data transmission over loss recovery, utilizing spare opportunities to conduct proactive loss recovery when there is currently no new data. As such, FUSO avoids sacrificing throughput to transmit redundant recovery packets ahead of new data, which would increase the FCT. Moreover, FUSO can dynamically adapt its redundancy rate to the network condition (Fig. 5(c) and Fig. 6(b)), by strictly following the congestion control constraint. Thus, FUSO naturally generates relatively more redundancy to accelerate loss recovery when the congestion

[7] There are only 3 layers in our simulation topology, thus we merge the portion of those lossy links *at* and *above* the 3^{rd} layer in the real topology into one layer in the simulated topology.



(a) Web-search: Avg FCT (b) Web-search: 99^{th} FCT (c) Data-mining: Avg FCT (d) Data-mining: 99^{th} FCT

Figure 8: Simulations under 10Gbps fat tree: Average and 99^{th} FCT (log scale) of latency-sensitive flows from web-search and data-mining workloads. Lossy links are randomly generated according to realistic measurements in §2.1.

is light, and becomes conservative when the congestion is heavy, which outperforms other methods' (*e.g.*, TCP-IR, Repflow, and Proactive) fixed redundancy rate. As such, FUSO's redundancy helps to greatly decrease the FCT, meanwhile only slightly affecting the overall network throughput (§5.2.2).

Although we focus on how FUSO improves the performance of small latency-sensitive flows in §5, FUSO is also applicable to long flows, which are typically bandwidth greedy and *cwnd* limited. Therefore, the necessary condition for proactive multi-path loss recovery in FUSO (when there is no more new data to be sent and the flow has spare *cwnd* slots) is only triggered at the end of the long flows. Previous studies [13] have shown that MPTCP provides a very good performance for long flows. We first ran experiments to compare FUSO and MPTCP, and the results (omitted for space limitation) confirm that FUSO incurs a negligible overhead and behaves almost exactly the same as the original MPTCP for long flows. We then enabled FUSO for all small and long flows and reran all experiments in §5, and observed that FUSO still outperforms other methods at the tail FCT. We also note that enabling MPTCP in general in data centers may hurt the average FCT of small flows for the following reason (also revealed in [36,41]): Originally designed for improving long flow's throughput, MPTCP's current congestion control can cause burstiness of multiple sub-flows and drive up queue lengths on all paths. How to improve multi-path congestion control, however, is orthogonal to FUSO and beyond the scope of this paper.

7 Related Work

Besides the works [4, 13–15, 22, 23, 31] that we have previously discussed in-depth, there is a rich literature on the general TCP loss recovery (*e.g.*, [18, 19, 32, 42]), short flows' tail FCT in both DCN (*e.g.*, [43–45]) and Internet (*e.g.*, [46, 47]), and utilizing multi-path in the transport (*e.g.*, [48–51]). Due to space limitation, we do not review these works in details. The key difference between FUSO and these works is that, to the best of our knowledge, FUSO is the first work to address the long tail FCT of short flows in DCN caused by failure-packet-loss-incurred timeout. FUSO is also the first systematic work to utilize multi-path diversity to conduct proactive transport loss recovery in DCN.

It is noteworthy that several data centers have recently deployed Remote Direct Memory Access (RDMA) [52, 53], a complementary technique to TCP. It relies on Priority-based Flow Control (PFC) to remove congestion drops. However, RDMA would perform badly in face of failure-induced loss (*e.g.*, even a slight 0.1%) due to its simple go-back-N loss recovery schemes [3]. FUSO is able to deal with both congestion-induced loss and failure-induced loss, and works for the widely used TCP in DCN [3, 6, 16]. We will study how to apply the principle of FUSO to RDMA/PFC in the future.

8 Conclusion

The chase for ultra-low FCT in data center networks has been a very active research area, and the solutions range from better topology and routing designs, optical switching, flow scheduling, congestion control, to protocol architectures (*e.g.*, RDMA/PFC), *etc.* This paper adds an important thread to this area, which is to properly leverage the inherent multi-path diversity for transport loss recovery, to deal with both failure-induced and congestion-induced packet loss in DCN. In our proposed FUSO, when a multi-path transport sender suspects loss on one sub-flow, recovery packets are immediately sent over another sub-flow that is not or less lossy *and* has spare congestion window slots. Our experiments show that the *fast* yet *cautious* FUSO can decrease the tail FCT by up to ∼82.3% (testbed) and ∼87.9% (simulation).

Acknowledgment

We thank Albert Greenberg, Ming Zhang, Jennifer Rexford, Fengyuan Ren, and Zhen Xiao for their valuable feedbacks on the paper's initial version, and Xiaohui Nie, Dapeng Liu, Kaixin Sui, and Hao Wang for their suggestions on improving this work. We thank Wei Bai for the help on the implementation and experiments, and Yousef Azzabi for the proofreading. We thank our shepherd Noa Zilberman and the conference reviewers for the helpful comments. This work was partly supported by the State Key Program of National Science of China under grant 61233007, the National Natural Science Foundation of China under grant 61472214 and 61472210, the National High Technology Development Program of China (863 program) under grant 2013AA013302, the National Key Basic Research Program of China (973 program) under grant 2013CB329105, the Tsinghua National Laboratory for Information Science and Technology key projects, and the China's Recruitment Program (Youth) of Global Experts.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74. ACM, 2010.

[2] Nandita Dukkipati and Nick McKeown. Why Flow-completion Time is the Right Metric for Congestion Control. *SIGCOMM Computer Communnication Review*, 36(1):59–62, January 2006.

[3] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 63–74. ACM, 2015.

[4] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 159–170. ACM, 2013.

[5] Yanpei Chen, Rean Griffith, Junda Liu, Randy H. Katz, and Anthony D. Joseph. Understanding TCP Incast Throughput Collapse in Datacenter Networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82. ACM, 2009.

[6] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hoelzle, Stephen Stuart, and Amin Vahdat. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Googles Datacenter Network. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 63–74. ACM, 2015.

[7] Haitao Wu, Jiabo Ju, Guohan Lu, Chuanxiong Guo, Yongqiang Xiong, and Yongguang Zhang. Tuning ECN for Data Center Networks. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, pages 25–36. ACM, 2012.

[8] Matt Calder, Rui Miao, Kyriakos Zarifis, Ethan Katz-Bassett, Minlan Yu, and Jitendra Padhye. Don'T Drop, Detour! In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 503–504. ACM, 2013.

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference*, SIGCOMM '08, pages 63–74. ACM, 2008.

[10] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference*, SIGCOMM '09, pages 51–62. ACM, 2009.

[11] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2009 Conference*, SIGCOMM '09, pages 63–74. ACM, 2009.

[12] Yuval Bachar. Disaggregation - The New Way to Build Mega (and Micro) Data Centers. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 1–1. IEEE, 2015.

[13] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277. ACM, 2011.

[14] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*, pages 29–29. USENIX Association, 2012.

[15] Alan Ford, Costin Raiciu, Mark Handley, and Olivier Bonaventure. *TCP extensions for multipath operation with multiple addresses*. Internet Engineering Task Force, January 2013. RFC 6824.

[16] Glenn Judd. Attaining the Promise and Avoiding the Pitfalls of TCP in the Datacenter. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, May 2015.

[17] M Allman, V Paxson, and E Blanton. *TCP congestion control*. Internet Engineering Task Force, September 2009. RFC 5681.

[18] E Blanton, M Allman, L Wang, I Jarvinen, M Kojo, and Y Nishida. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. Internet Engineering Task Force, 2012. RFC 6675.

[19] M Allman, K Avrachenkov, U Ayesta, J Blanton, and P Hurtig. *Early retransmit for TCP and stream control transmission protocol (SCTP)*. Internet Engineering Task Force, April 2010. RFC 5827.

[20] Netem page from Linux foundation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[21] Stephen Hemminger et al. Network emulation with NetEm. In *Linux conf au*, pages 18–23. Citeseer, 2005.

[22] N Dukkipati, N Cardwell, Y Cheng, and M Mathis. *Tail loss probe (TLP): An algorithm for fast recovery of tail losses*. Internet Engineering Task Force, April 2013. RFC 5827.

[23] Tobias Flach, Yuchung Cheng, Barath Raghavan, and Nandita Dukkipati. *TCP Instant Recovery: Incorporating Forward Error Correction in TCP*. Internet Engineering Task Force, April 2013. RFC 5827.

[24] Peng Cheng, Fengyuan Ren, Ran Shu, and Chuang Lin. Catch the Whole Lot in an Action: Rapid Precise Packet Loss Notification in Data Center. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, April 2014.

[25] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, pages 19–19. USENIX Association, 2010.

[26] Christian E Hopps. *Analysis of an equal-cost multi-path algorithm*. Internet Engineering Task Force, 2000. RFC 2992.

[27] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. Explicit path control in commodity data centers: Design and applications. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, May 2015.

[28] S Floyd, J Mahdavi, M Mathis, and A Romanow. *TCP Selective Acknowledgment Options*. Internet Engineering Task Force, 1996. RFC 2018.

[29] Christoph Paasch, Gregory Detal, Kenjiro Nakayama, Gucea Doru, Matthieu Baerts, Jaehyun Hwang, Sbastien Barr, and et al. MPTCP Linux kernel implementation. *git://github.com/multipath-tcp/mptcp*.

[30] The Network Simulator - ns-2. *http://www.isi.edu/nsnam/ns/*.

[31] Hong Xu and Baochun Li. RepFlow: Minimizing Flow Completion Times with Replicated Flows in Data Centers. In *Proceedings of the IEEE INFOCOM 2014 Conference*, pages 1581–1589, April 2014.

[32] Sally Floyd, Hari Balakrishnan, and Mark Allman. *Enhancing TCP's loss recovery using limited transmit*. Internet Engineering Task Force, January 2001. RFC 3042.

[33] Net-tcp-fec: Modifications to the Linux networking stack to enable forward error correction in TCP. *http://tflach.github.io/net-tcp-fec/*.

[34] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*, March 2016.

[35] Edgar N Gilbert. Capacity of a Burst-Noise Channel. *Bell system technical journal*, 39(5):1253–1265, 1960.

[36] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *Proceedings of the ACM SIGCOMM 2014 Conference*, SIGCOMM '14, pages 503–514. ACM, 2014.

[37] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[38] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never Than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61. ACM, 2011.

[39] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 Conference*, SIGCOMM '09, pages 303–314. ACM, 2009.

[40] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 435–446. ACM, 2013.

[41] Jiaxin Cao, Rui Xia, Pengkun Yang, Chuanxiong Guo, Guohan Lu, Lihua Yuan, Yixin Zheng, Haitao Wu, Yongqiang Xiong, and Dave Maltz. Per-packet Load-balanced, Low-latency Routing for Clos-based Data Center Networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 49–60. ACM, 2013.

[42] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 281–291. ACM, 1996.

[43] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 139–150. ACM, 2012.

[44] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). In *Proceedings of the ACM SIGCOMM 2012 Conference*, SIGCOMM '12, pages 115–126. ACM, 2012.

[45] Matthew P Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert NM Watson, Andrew W Moore, Steven Hand, and Jon Crowcroft. Queues Dont Matter When You Can JUMP Them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*, pages 1–14, 2015.

[46] Qingxi Li, Mo Dong, and P Brighten Godfrey. Halfback: Running Short Flows Quickly and Safely. In *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15. ACM, 2015.

[47] Jianer Zhou, Qinghua Wu, Zhenyu Li, Steve Uhlig, Peter Steenkiste, Jian Chen, and Gaogang Xie. Demystifying and Mitigating TCP Stalls at the Server Side. In *Proceedings of the 11th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15. ACM, 2015.

[48] Ming Zhang, Junwen Lai, Arvind Krishnamurthy, Larry L Peterson, and Randolph Y Wang. A Transport Layer Approach for Improving End-to-End Performance and Robustness Using Redundant Paths. In *USENIX Annual Technical Conference, General Track*, pages 99–112, 2004.

[49] Peter Key, Laurent Massoulié, and Don Towsley. Path Selection and Multipath Congestion Control. In *Proceedings of the IEEE INFOCOM 2007 Conference*, pages 143–151. IEEE, 2007.

[50] Yong Cui, Lian Wang, Xin Wang, Hongyi Wang, and Yining Wang. FMTCP: A Fountain Code-based Multipath Transmission Control Protocol. *Networking, IEEE/ACM Transactions on*, 23(2):465–478, 2015.

[51] Morteza Kheirkhah, Ian Wakeman, and George Parisis. MMPTCP: A Multipath Transport Protocol for Data Centers. In *Proceedings of the IEEE INFOCOM 2016 Conference*, April 2016.

[52] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 523–536. ACM, 2015.

[53] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based Congestion Control for the Datacenter. In *Proceedings of the ACM SIGCOMM 2015 Conference*, SIGCOMM '15, pages 537–550. ACM, 2015.

StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs

Kenichi Yasukata[†1], Michio Honda[2], Douglas Santry[2], and Lars Eggert[2]
[1]*Keio University*
[2]*NetApp*

Abstract

StackMap leverages the best aspects of kernel-bypass networking into a new low-latency Linux network service based on the full-featured TCP kernel implementation, by dedicating network interfaces to applications and offering an extended version of the netmap API as a zero-copy, low-overhead *data path* while retaining the socket API for the *control path*. For small-message, transactional workloads, StackMap outperforms baseline Linux by 4 to 80 % in latency and 4 to 391 % in throughput. It also achieves comparable performance with Seastar, a highly-optimized user-level TCP/IP stack for DPDK.

1 Introduction

The TCP/IP protocols are typically implemented as part of an operating system (OS) kernel and exposed to applications through an application programming interface (API) such as the socket API [61] standard. This protects and isolates applications from one another and allows the OS to arbitrate access to network resources. Applications can focus on implementing their specific higher-level functionality and need not deal with the details of network communication.

A shared kernel implementation of TCP/IP has other advantages. The commercialization of the Internet has required continuous improvements to end-to-end data transfers. A collaboration between commercial and open source developers, researchers and IETF participants over at least the last 25 years has been improving TCP/IP to scale to increasingly diverse network characteristics [11, 39, 58], growing traffic volumes [13, 32], and improved tolerance to throughput fluctuations and reduced transmission latencies [1, 10, 49].

A modern TCP/IP stack is consequently a complex, highly optimized and analyzed piece of software. Due to these complexities, only a small number of stacks (*e.g.,* Linux, Windows, Apple, BSD) have a competitive feature set and performance, and therefore push the vast majority of traffic. Because of this relatively small number of OS stacks (compared to the number of applications), TCP/IP improvements have a well-understood and relatively easy deployment path via kernel updates, without the need to change applications.

However, implementing TCP/IP in the kernel also has downsides, which are becoming more pronounced with larger network capacities and applications that are more sensitive to latency and jitter. Kernel data processing and queueing delays now dominate end-to-end latencies, particularly over uncongested network paths. For example, the fabric latency across a datacenter network is typically only a few µs. But a minimal HTTP transaction over the same fabric, consisting of a short "GET" request and an "OK" reply, takes tens to hundreds of µs (see Section 3).

Several recent proposals attempt to avoid these overheads in a radical fashion: they bypass the kernel stack and instead implement all TCP/IP processing inside the application in user space [24, 29, 37] or in a virtual machine context [4]. Although successful in avoiding overheads, these kernel-bypass proposals also do away with many of the benefits of a shared TCP/IP implementation: They usually implement a simplistic flavor of TCP/IP that does not include many of the performance optimizations of the OS stacks, it is unclear if and by whom future protocol improvements would be implemented and deployed, and the different TCP/IP versions used by different applications may negatively impact one another in the network.

It is questionable whether kernel-bypass approaches are suitable even for highly specialized network environments such as datacenters. Due to economic reasons [17], they are assembled from commodity switches and do not feature a centralized flow scheduler [2, 45]. Therefore, path characteristics in such datacenters vary, and more advanced TCP protocol features may be useful in order to guarantee sub-millisecond flow completion times.

[†]Most of the research was done during an internship at NetApp.

Another group of recent proposals [16, 22, 46] attempts to reduce the overheads associated with using the kernel stack by optimizing the socket API in a variety of ways. Unlike kernel-bypass approaches, which dedicate network interfaces (NICs) to individual applications, they continue to allow different applications to share NICs. This limits the benefit of this set of proposals, due to the continued need for kernel mechanisms to arbitrate NIC access, and motivates our StackMap proposal.

On today's multi-core and multi-NIC servers, it is common practice to dedicate individual cores and individual (virtual) interfaces to individual applications; even more so for distributed scale-out applications that use all system resources across many systems.

This paper presents StackMap, a new OS network service that dedicates NICs to individual applications (similar to kernel-bypass approaches) but continues to use the full-fledged kernel TCP/IP stack (similar to API-optimizing approaches). StackMap establishes low-latency, zero-copy data paths from dedicated NICs, through the kernel TCP/IP implementation, across an extended version of the netmap API into StackMap-aware applications. The extended netmap API features operations for explicit execution of packet I/O and TCP/IP processing, event multiplexing and application-data I/O, which is tightly coupled with an abstraction of the NIC packet buffers. Alongside the data path, StackMap also retains the regular socket API as a control path, which allows sharing of the OS network stack properly with regular applications.

StackMap outperforms Linux by 4 to 80 % in average latency, 2 to 70 % in 99th-percentile latency and 4 to 391 % in throughput. StackMap also improves memcached throughput by 42 to 133 %, and average latency by 30 to 58 %. Up to six CPU cores, StackMap even outperforms memcached on Seastar, which is a highly-optimized, user-level TCP/IP stack for DPDK.

The StackMap architecture resembles existing kernel-bypass proposals [4, 6] and borrows common techniques from them, such as batching, lightweight buffer management and the introduction of new APIs. However, applying these techniques to an OS network stack requires a significant design effort, because of its shared nature. Additionally, the kernel TCP/IP implementation depends heavily on other components of the network stack and other kernel subsystems, such as the memory allocator, packet I/O subsystem and socket API layer. StackMap carefully decomposes the OS network stack into its TCP/IP implementation and other components, and optimizes the latter for transactional workloads over dedicated NICs.

StackMap inherits the complete set of the rich TCP/IP protocol features from the OS stack. A latency analysis of the Linux kernel implementation shows that TCP/IP processing for both the transmit and receive directions consumes less than 0.8 µs, respectively. This is less than the half the overall request handling latency of 3.75 µs (see Section 3.1).

In summary, we make three major contributions: (1) a latency analysis of the Linux kernel TCP/IP implementation, (2) the design and implementation of StackMap, a new low-latency OS networking service, which utilizes dedicated NICs together with the kernel TCP/IP implementation, and (3) a performance evaluation of StackMap and a comparison against a kernel-bypass approach.

The remainder of this paper is organized as follows: Section 2 discusses the most relevant prior work, much of which StackMap draws motivation from. Section 3 analyses latencies of the Linux network stack, in order to validate the feasibility of the StackMap approach and identify problems we must address. Section 4 describes the design and implementation of StackMap. Section 5 evaluates StackMap. Section 6 discusses the inherent limitations posed by the StackMap architecture as well as those only present in the current prototype implementation. The paper concludes with Section 7.

2 Motivation and Related Work

Inefficiencies in OS network stacks are a well-studied topic. This section briefly discusses the most relevant prior work in the field from which StackMap draws motivation, and summarizes other related work.

2.1 Kernel-Bypass Networking

An OS typically stores network payloads in kernel-space buffers, accompanied by a sizable amount of metadata (*e.g.*, pointers to NICs, socket and protocol headers). These buffers are dynamically allocated and managed by reference counts, so that producers (*e.g.*, a NIC driver in interrupt context) and consumers (*e.g.*, `read()` in syscall context) can operate on them [55].

Around 2010–2012, researchers developed approaches to use static, pre-allocated buffers with a minimum amount of metadata for common, simple packet operations, such as packet forwarding after IPv4 longest-prefix matching [62]. A key architectural feature of these systems [21, 54] is to perform all packet processing in user space[1], by moving packets directly to and from the NIC, bypassing the OS stack. These systems also extensively exploit batching, *e.g.*, for syscalls and device access, which is preceded by a write barrier. The outcome of these research proposals is general frameworks for fast, user-space packet I/O, such as netmap [54] and DPDK [26].

[1] Click [43], proposed in 1999, has an option to run in user-space, but as a low-performance alternative to the in-kernel default, due to using the traditional packet I/O API or Berkeley Packet Filter. A netmap-enabled version of Click from 2012 outperforms the in-kernel version [52].

In 2013–2014, "kernel-bypass" TCP stacks based on these user-space networking frameworks emerged, driven by a desire to further improve the performance of transactional workloads. UTCP [24] and Sandstorm [37] build on netmap; mTCP [29] on PacketShader; Seastar [6], IX [4] and UNS [27] on DPDK. They either re-implement TCP/IP almost from scratch or rely on existing—often similarly limited or outdated—user-space stacks such as lwIP [12], mostly because of assumed inefficiencies in OS stacks and reported inefficiencies of running more modern OS stack code in user-space via shims [31, 60].

Further, these kernel-bypass stacks introduce new APIs to avoid inefficiencies in the socket API, shared-nothing designs to avoid locks, active NIC polling to avoid handling interrupts, network stack processing in application threads to avoid context switches and synchronization, and/or direct access to NIC packet buffers. These techniques become feasible, because kernel-bypass approaches dedicate NICs or packet I/O primitives, such as a NIC rings, to application threads. In other words, they do not support sharing NICs with applications that use the OS stack. Each thread therefore executes its own network stack instance, and handles all packet I/O, network protocol processing in addition to executing the application logic.

However, such techniques are not inherently limited to user-space use. In 2015, mSwitch [23] demonstrated that they can be incorporated into kernel code (with small modifications) and result in similar benefits: mSwitch speeds up the Open vSwitch data path by a factor of three. It retains the OS packet representation, but simplifies allocation and deallocation procedures by performing all packet forwarding operations within the scope of a single function. StackMap borrows the idea of using acceleration techniques first used for kernel-bypass networking inside the kernel from mSwitch.

2.2 Network API Enhancements

Server applications typically execute an event loop to monitor file descriptors, including network connections. When an event multiplexing syscall such as `epoll_wait()` returns "ready" file descriptors, the application iterates over them, *e.g.*, to read requests, send responses, or to accept new connections.

In 2012, MegaPipe [22] improved on this common pattern by introducing a new API that featured two techniques: First, it batches syscall processing across multiple file descriptors (similarly to FlexSC [59]) to amortize their cost over a larger number of bytes. This is particularly effective for transactional workloads with small messages. Second, it introduces new *lightweight sockets*, which relax some semantics of regular sockets that limit multi-core scalability, such as assigning the lowest available integer to number a new descriptor (which requires a global lock for the entire process). Not surprisingly, this approach provides a smaller performance improvement compared to the kernel-bypass approaches that dedicate NICs to applications [29].

2.3 TCP Maintenance

Originally, TCP was a relatively simple protocol [48] with few extensions [28, 38], and the implementation and maintenance cost was very manageable. In addition to implementations by general-purpose OS vendors, many more specialized TCP implementations were undertaken to support purpose-specific appliances, such as middleboxes [25, 40], storage servers [15] and embedded systems [34].

Over the years, maintaining a TCP implementation has become much more challenging. TCP is being improved at a more rapid pace than ever, in terms of performance [1, 3, 39, 49, 57], security [5, 13, 32, 44, 51], as well as more substantial extensions such as Multipath TCP [18]. In addition to the sheer number of TCP improvements that stacks need to implement in order to remain competitive in terms of performance, security and features, the improvements themselves are becoming more complex. They need to take the realities of the modern Internet into account, such as the need for correct operation in the presence of a wide variety of middleboxes or the scarcity of available option space in the TCP header [5, 49, 50].

Consequently, the set of "modern" TCP stacks that offer the best currently achievable performance, security and features has been shrinking, and at the moment consists mostly of the large general-purpose OS stacks with a sizable developer base and interest. Many other TCP implementations have fallen behind, and it is very uncertain whether they will ever catch up. This is especially regrettable for stacks that underlie many deployed middleboxes, because they in turn limit future Internet evolution [25, 40]. The situation is unfortunately similar for many of the recent kernel-bypass stacks (see Section 2.1), none of which shows signs of very active maintenance. Networking products that wish to take advantage of the performance benefits of kernel-bypass solutions thus run the risk of adopting a stack that already is not competitive in terms of features, and may not not remain competitive in terms of performance in the future.

StackMap mitigates this risk, by offering similar performance benefits to kernel-bypass approaches while using the OS stack, which has proven to see active maintenance and regular updates.

2.4 Other Related Work

Some other pieces of relevant related work exist, in addition to the general areas discussed above.

In the area of latency analyses of network stacks, [33] analyses the latency breakdown of memcached, in order to

understand how to fulfill the quality-of-service objectives of multiple workloads. Their measurements show much higher processing delays than ours, and their focus is not on improving the OS network service. Also, [4] reports a one-way latency of 24 µs with Linux TCP/IP, which is twice what we measure in Section 3. Our results are similar to those reported in [47], taking into account that they use UDP instead of TCP.

In the area of API extensions, many UNIX variants (including Linux) implement a `sendfile()` syscall that transmits a regular file from the buffer cache directly into a TCP socket without a memory copy or multiple context switches. The Linux `sendmmsg()` and `recvmmsg()` syscalls support passing multiple messages to and from the kernel at a time. However, they do not allow batching across different descriptors. In late 2015, the Linux kernel connection multiplexer (KCM) [7] was proposed. It enables applications to use message-based interfaces, such as `sendmmsg()`, over TCP and allows syscall batching across multiple TCP connections. Linux busy poll sockets [8] permit to directly poll a network device to avoid interrupts when receiving packets. Windows IOCP [42], the Linux `epoll` and BSD `kqueue` families of syscalls are event multiplexing APIs. All of these approaches are limited by needing to remain compatible with the semantics established by the socket API and to arbitrate NIC access, which incurs significant overheads (see Section 3).

Kernel-bypass network stacks introduce other techniques to optimize some TCP code paths, such as sorting TCP connections by timeout order or pre-allocating TCP protocol control blocks for fast connection setup. None of these techniques are inherently limited to kernel-bypass approaches, and StackMap will immediately gain their benefits once the kernel stack implements them. The same is true for Fastsocket [35], a recent optimization of the Linux stack for multi-core scalability—when Fastsocket functionality is present in the kernel, applications using StackMap will immediately gain its benefits.

3 Design Space Analysis

Unless a network hop becomes the bottleneck, the end-to-end latency of a transactional workload depends on two main factors: (1) the processing delays of the network stack and the application, and (2) the queueing latency, particularly in the presence of concurrent requests. This section analyzes these latency factors for the Linux kernel network stack, in order to determine the feasibility of using the kernel TCP/IP implementation for low-latency networking and to identify any challenges such an architecture must address.

Layer	Component	Time [µs]
Kernel	Driver RX	0.60
	Ethernet & IPv4 RX	0.19
	TCP RX	0.53
	Socket enqueue	0.06
Application	`epoll_wait()` syscall	0.15
	`read()` syscall	0.33
	Generate "OK" reply	0.48
	`write()` syscall	0.22
Kernel	TCP TX	0.70
	IPv4 & Ethernet TX	0.06
	Driver TX	0.43
Total		3.75

Table 1: Request processing overheads at a server.

3.1 TCP/IP Processing Cost

We start by analyzing a single, short-message request-response exchange (96 B "GET", 127 B "OK") between two Linux machines connected with 10 G Ethernet NICs (see Section 5 for configuration details). We use Systemtap [14] to measure processing delays in particular components of the server.

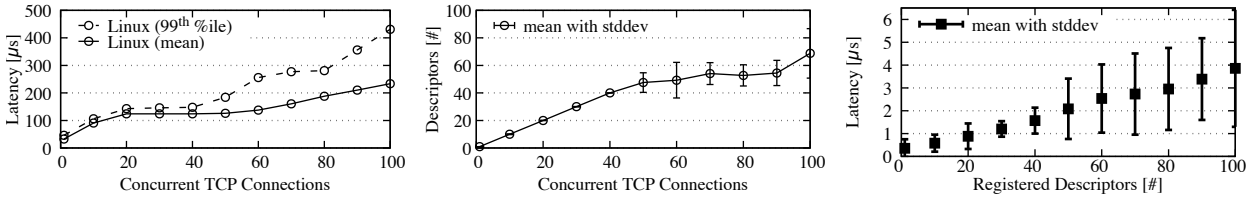
Table 1 shows the processing overheads measured at the various layers during this experiment. The key insight is that TCP/IP processing takes 0.72 µs on receive (Ethernet & IPv4 RX and TCP RX) and 0.76 µs on transmit (TCP TX and IPv4 & Ethernet TX). The combined overhead of 1.48 µs is not a large factor of the overall processing delay of 3.75 µs, and of the end-to-end one-way latency of 9.75 µs (half of the round-trip latency reported by `wrk`).

The significant difference of 6 µs between the end-to-end one-way latency and the processing delay of the server is due to link, PCIe bus and switch latencies (1.15 µs combined, one-way), and some indirection between the hardware and software, which is unavoidable even for kernel-bypass TCP/IPs. We confirm this finding by running a netmap-based ping-pong application between the same machines and NICs, which avoids most of the network-stack and application processing. The result is a one-way latency of 5.77 µs, which is reasonably similar to the 6 µs measured before.

Data copies do not appear to cause major overheads for short-message transactions. In this experiment, copying data only takes 0.01 and 0.06 µs for 127 and 1408 B of data, respectively (not shown in Table 1).

3.2 Latencies for Concurrent Connections

A busy server continually serves a large number of requests on many different TCP connections; with clients using potentially multiple parallel connections to the server to avoid head-of-line blocking. For new data arriving on connec-



(a) End-to-end latency. (b) Descriptors returned by `epoll_wait()`. (c) Processing delay of `epoll_wait()`.

Figure 1: Latency analysis of concurrent TCP connections.

tions (*i.e.*, new client requests), the kernel needs to mark the corresponding file descriptor as "ready". Applications typically run an event loop on each CPU core around an event multiplexing syscall such as `epoll_wait()` to identify any "ready" connections and serve the new batch of requests that has arrived on them. During application processing of one such batch of requests, queuing delays can occur in two ways: First, an application iterates over ready descriptors one after the other, serving their requests. When many requests arrive on many connections, requests on connections iterated over at the end of the cycle incur significant delays. Second, new requests arriving while an application is processing the current batch are queued in the kernel until the next cycle. This behavior increases both the mean and tail end-to-end latencies during such an event processing cycle, proportional to the number of descriptors processed during the cycle.

In order to demonstrate this effect and quantify its latency impact, we repeat the measurement from Section 3.1, but generate concurrent requests on several TCP connections. Figure 1a plots the mean (solid lines) and 99th-percentile (dashed lines) end-to-end latencies, as measured by the client. Figure 1b shows the measured mean number of descriptors returned by `epoll_wait()` at the server (with standard deviations). There is a clear correlation between the end-to-end latencies and the number of file descriptors returned, as the number of concurrent connections increases.

Concurrent connections also increase the processing delay of the `epoll_wait()` syscall, which iterates over all the descriptors registered in the kernel. Figure 1c quantifies this cost for different numbers of registered descriptors, using Systemtap to measure time spent in `sp_send_events_proc()`. The cost to iterate over ready descriptors is negligible when the number of registered descriptors is small. However, it reaches about 1 µs for 20 descriptors, and almost 4 µs for 100 descriptors, which is substantial.

Note that the processing delay for one registered descriptor in Figure 1c is slightly higher than that reported for `epoll_wait()` in Table 1, because the numbers reported there subtract the Systemtap overhead from the result (estimated by measuring the extremely cheap (`tcp_rcv_space_adjust()` function).

3.3 Takeaway

It is clear that compatibility with the socket API comes at a significant cost, including the overheads of `read()` and `write()` (Table 1) as well as `epoll_wait()` (Figure 1c). Packet I/O also introduces significant overheads (Driver TX and RX in Table 1), and more performance is lost due to the inability of batching transmissions over multiple TCP connections. With concurrent TCP connections, the overheads associated with request processing result in long average and tail latencies due to queueing delays, on the order of tens to hundreds of µs (Figure 1a).

4 StackMap Design

The discussion in the last two sections leads us to three starting points. First, there are a number of existing techniques to improve network stack efficiency. StackMap should incorporate as many of them as possible. Second, it must use an actively-maintained, modern TCP/IP implementation, *i.e.*, one of the main server OS stacks. And StackMap must use this stack in a way that lets it immediately benefit from future improvements to that code, without the need to manually port source code changes. This is important, so that applications using StackMap are not stuck with an outdated stack. Finally, while TCP/IP protocol processing in an OS stack is relatively cheap, StackMap must improve other overheads, most notably ones related to the API and packet I/O, in order to significantly reduce queueing latency in the presence of concurrent TCP connections.

4.1 StackMap Design Principles

StackMap dedicates NICs to privileged applications through a new API. We believe this is a reasonable principle for today's high-performance systems, and the same approach is already followed by kernel-bypass approaches (*netmap, DPDK*). However, unlike such kernel-bypass approaches, StackMap also "maps" the dedicated NICs into the kernel TCP/IP stack. This key differentiator results in key benefits, because many overheads of the traditional socket API and buffer management can be avoided.

A second design principle is retaining isolation: although StackMap assumes that applications are privileged (they see all traffic on dedicated NICs), it must still protect the OS and any other applications when a privileged StackMap application crashes. StackMap inherits most of its protection mechanisms from netmap, which it is based on, including protection of NIC registers and exclusive ring buffer operations between the kernel and user space. We discuss the limitations posed by our current prototype in Section 6.

A third design principle is backwards compatibility: when a NIC is dedicated to a StackMap application, regular applications must remain able to use other NICs. StackMap achieves this by retaining part of the socket API for control plane operations. Since today's commodity OSes mostly use monolithic kernels, StackMap must share a single network stack instance across all NICs, whether they are dedicated to privileged applications or shared by regular applications. One implication of this design principle is that it makes a complete shared-nothing design difficult, *i.e.*, some coordination remains required. However, Section 5 shows that this coordination overhead is small. Additionally, the OS stack is increasingly being disaggregated into shared objects, such as accept queues, and StackMap will benefit from such improvements directly, further improving future performance.

4.2 StackMap Architecture

The StackMap design centers around combining a fast packet I/O framework with the OS TCP/IP stack, to give application fast message-oriented communication over TCP connections, which has been crucial for the applications like memcached, web servers and content delivery network (CDN) servers, to name a few [7, 22]. Thus, in addition to dedicating NICs to privileged applications, StackMap must also enable the kernel stack to apply its regular TCP/IP processing to those NICs. To this end, StackMap extends the netmap framework to allow it to efficiently integrate with the OS stack.

DPDK is not a suitable basis for StackMap, because it executes its NIC drivers entirely in user space. It is difficult to efficiently have such user-space NIC drivers call into the kernel network stack.

Although netmap already supports communicating with the OS stack [54], its current method has significant overheads, because it is unoptimized and only focuses on applications that use the socket API, which as we have shown to have undesirable overheads.

Figure 2 illustrates the StackMap architecture. StackMap (i) mediates traffic between a dedicated NIC and a privileged application through a slightly extended version of the netmap API; (ii) uses the kernel TCP/IP stack to process incoming TCP packets and send outgoing

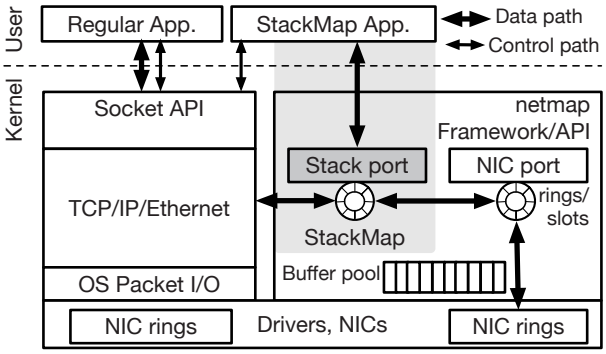


Figure 2: StackMap architecture overview.

application data; (iii) and uses the regular socket API for control, to both share the protocol/port-number space with regular applications and to present a consistent picture of clients and connections to the kernel.

For the data path API, the goal of StackMap is to combine the efficiency, generality and security of the netmap packet I/O framework with the full-fledged kernel TCP/IP implementation, to give applications a way to send and receive messages over TCP with much lower overheads compared to the socket API.

These performance benefits will require an application to more carefully manage and segment the data it is handling. However, we believe that this is an acceptable trade-off, at least for transactional applications that care about message latencies rather than bulk throughput. In addition, because StackMap retains the socket API for control purposes, an application can also still use the standard data plane syscalls *e.g.*, `read()` and `write()` (with the usual associated overheads).

4.3 Netmap Overview

Netmap [54] maintains several *pools* of uniquely-indexed, pre-allocated *packet buffers* inside the kernel. Some of these buffers are referenced by *slots*, which are contained in *rings*. A set of rings forms a *port*. A NIC port maps its rings to NIC hardware rings for direct packet buffer access (Figure 2). A *pipe* port provides a zero-copy point-to-point IPC channel [53]; a *VALE* port is a virtual NIC of a VALE/mSwitch [23, 56] software switch instance (not shown in Figure 2).

The netmap *API* provides a common interface to all types of ports. It defines methods to manipulate rings and uses `poll()` and `ioctl()` syscalls for synchronization with the kernel, whose backend performs port-specific operations, *e.g.*, device I/O for NIC ports or packet forwarding for VALE ports. Netmap buffers that are part of the same pool are interchangeable between slots, even across different rings or ports, which enables zero-copy operations.

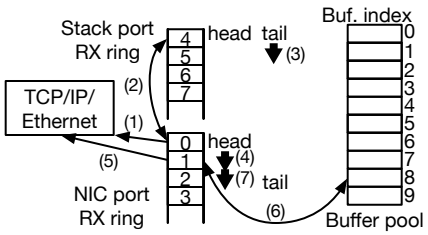


Figure 3: TCP in-order and out-of-order reception. Initially, buffers 0–7 are linked to either of the rings, and 8–9 are extra buffers.

When a netmap client (*i.e.*, an application) registers a port, netmap exposes the corresponding rings by mapping them into its address space. Rings can be manipulated via *head* and *tail* pointers, which indicate the beginning and end of the user-owned region. A client fills a TX ring with new packets from head and advances it accordingly. On the next synchronization syscall, netmap transmits packets from the original head to the current head. During a packet reception syscall, netmap places new packets onto the ring between head and tail; the client then consumes these packets by traversing the ring from head to tail, again advancing the head accordingly.

4.4 StackMap Data Path

StackMap implements a new type of netmap port, called a *stack port*. A stack port runs in cooperation with a NIC port, and allocates packet buffers for its rings from the same pool. StackMap also uses buffers that are not linked to any slot in the same pool as the NIC and stack port. These buffers allow StackMap to retain packets in netmap buffers, but outside NIC or the stack port rings, which prevents them from being processed by ring operations. This is useful, for example, to store packets that may require retransmission or to maintain data that was received out-of-order. To utilize multiple CPU cores efficiently, for each core, one stack port ring and NIC port ring should be configured.

A ring belonging to a stack port can handle multiple TCP connections, which are exposed to the application through regular descriptors. This is essential for syscall *and* packet I/O batching over these connections. Thus, an application (on TX) or the kernel (on RX) indicates a corresponding connection or file descriptor for each slot. We later explain how this can be used by an application to process an entire RX ring for a particular descriptor, rather than looking up it for every packet.

Stack ports implement their own netmap syscall backend, so they can provide the netmap API to an application. Their TX and RX backends start StackMap data path processing. When a StackMap application performs a syscall for RX, the stack port backend first brings new packets into an RX ring of its NIC port, then instructs the OS stack to run them through its RX path. StackMap then moves any packets that the OS stack identifies as in-order TCP segments for their respective connections into an RX ring of its stack port. Out-of-order TCP segments are moved into the extra buffer space; They are delivered to the stack port RX ring when the out-of-order hole is filled. All buffer movements are performed by swapping indices, *i.e.*, zero-copy. Figure 3 illustrates these steps using two packets, where the second one arrives out-of-order. Step (1) and (5) process a packet in the TCP/IP stack. StackMap acts as a netmap "client" for a NIC port, thereby advancing the head pointer of its RX ring to consume packets. Conversely, StackMap acts as a netmap "backend" for a stack port. It thus advances the tail pointer of its RX ring when it puts new data into buffers.

The TX path is the opposite. When an application wishes to transmit new data located in buffers of a stack port TX ring, StackMap pushes these buffers through the TX path of the OS stack (see Table 1 for the overheads of those steps; in TCP TX StackMap skips packet buffer allocation, and so is somewhat faster). StackMap then intercepts the same packets after the Ethernet TX processing and moves the buffers into a TX ring of the NIC port (again zero-copy by swapping buffer indices). StackMap then advances the head of the NIC port ring and the old head of the stack port ring and triggers the NIC port for transmission. Since the OS stack would normally keep these TCP packets in its retransmission queue, StackMap unlinks the packet buffers after transmission.

If the size of data in a given TCP connection exceeds the available window of the respective connection *i.e.,* the minimum of the advertised receive window and the congestion window computed by the TCP stack, StackMap swaps the excess buffers out of the stack port TX ring, in order to avoid stalling the transmission on other connections. Any such buffers are moved back to the NIC port ring during the next operation.

To pass netmap buffers to the OS stack, a stack port pre-allocates a *persistent* `sk_buff` for each netmap buffer, which is the internal OS packet representation structure. This approach allows StackMap to avoid dynamic allocation and deallocation of `sk_buff`s, which has been shown to significantly reduce packet processing overheads [23].

In addition to being clocked by the stream of inbound acknowledgments (ACKs), the TCP protocol also has some inherent timers, *e.g.*, the retransmission timeout (RTO). StackMap processes any pending TCP timer events only on the TX or RX syscalls. This preserves the synchronization model of the netmap API between the kernel and user space, protecting buffers from concurrent access by a timer handler and user space code.

```
1  struct sockaddr_in sin = { AF_INET, "10.0.0.1", INADDR_ANY };
2  int sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
3  bind(sd, &sin);
4  // prefix "stack" opens stack port for given interface
5  struct nm_desc *nmd = nm_open("stack:ix0");
6  connect(sd, dst_addr); /* non-blocking */
7  // transmit using ring 0 only, for this example
8  struct netmap_ring *ring = NETMAP_TXRING(nmd->nifp, 0);
9  uint32_t cur = ring->cur;
10 while (app_has_data && cur != ring->tail) {
11     struct netmap_slot *slot = &ring->slot[cur];
12     char *buf = STACKMAP_BUF(ring, slot->buf_index);
13     // place payload in buf, then
14     slot->fd = sd;
15     cur = nm_ring_next(ring, cur);
16 }
17 ring->head = ring->cur = cur;
18 ioctl(nmd->fd, NIOCTXSYNC);
```

Figure 4: Initiating a connection and sending data.

4.5 StackMap API

In order to share the kernel TCP/IP stack with regular applications, StackMap retains the socket API for control, including, *e.g.*, the `socket()`, `bind()`, `listen()` and `accept()` syscalls. To reduce connection setup costs, StackMap optionally can perform `accept()` in the kernel before returning to user space, similar to MegaPipe [22].

The StackMap data path API has been designed to resemble the netmap API, except for a few extensions. One is the `STACKMAP_BUF(slot_idx, ring)` macro, which extends the netmap `NETMAP_BUF(slot_idx, ring)` macro and returns a pointer to the beginning of the payload data in a packet buffer. `STACKMAP_BUF` allows an application to easily write and read payload data to and from buffers, skipping the packet headers (which on TX are filled in by the kernel).

On TX, the application must indicate the descriptor for each slot to be transmitted, so that the OS stack can identify the respective TCP connections. On RX, the OS stack marks the slots accordingly, so that the application can identify which connection the data belongs to. Figure 4 illustrates use of the StackMap API for opening a new TCP connection and sending data in C-like pseudo code.

On RX, an application can consume data by simply traversing the RX ring of a stack port. However, this simple approach often does not integrate naturally into existing applications, because they are written to iterate over descriptors or connections, rather than iterating over data in packet arrival order. Unfortunately, using the `epoll_wait()` syscall is not an option because of significant overheads shown in Section 3.2.

StackMap thus introduces a new API that allows applications to consume data more naturally, ordered by descriptor. It is based on constructing a list of ready file descriptors during network stack processing, as well as grouping buffers for each descriptor, and by exploiting the opportunity that the application *synchronously* calls into the kernel network stack.

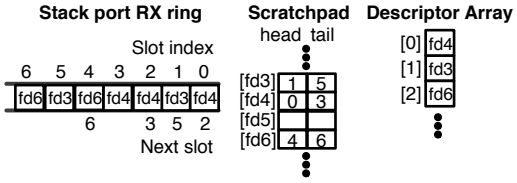


Figure 5: Algorithm to build an array of ready descriptors.

Figure 5 illustrates this algorithm with an example. Here, the OS stack has identified packets 0, 2 and 3 to belong to file descriptor 4 (`fd4`), packets 1 and 5 belong to `fd3`, and packets 4 and 6 to `fd6`. Note that descriptors are guaranteed to be unique per process. Each slot maintains a "next slot" index that points to the next slot of the same descriptor. StackMap uses a *scratchpad* table indexed by descriptor to maintain the head and tail slot index for each. The tail is used by the OS stack to append new data to a particular descriptor, by setting the "next slot" of the last packet to this descriptor, without having to traverse the ring. The head is used by the application to find the first packet for a particular descriptor, also without having to traverse the ring. This algorithm is inspired by mSwitch [23], and we expect similarly high performance and scalability.

The scratchpad is a process-wide data structure which requires 32 B (for two 16 B buffer indices) per entry, and usually holds 1024 entries (the default per-process maximum number of descriptors in Linux). We do not consider the resulting size of 32 KB problematic on today's systems even if it was extended by one or two orders of magnitude, but it would be possible to reduce this further by dynamically managing the scratchpad (which would incur some modest overhead).

When the first data for a particular descriptor is appended, StackMap also places the descriptor into a *descriptor array* (see Figure 5) that is exposed to the application. The application uses this array very similarly to how it would use the array returned by `epoll_wait()`, but without incurring the overhead of that syscall. Figure 6 illustrates how an application receives data and traverses the RX ring by descriptor.

The current API allows an application to traverse an RX ring both in packet-arrival order (*i.e.*, without using the descriptor array) and in descriptor order. In the future, StackMap may sort buffers in descriptor order when moving them into the stack port RX ring. This would remove the ability to traverse in packet order, but greatly simplifies the API and eliminates the need for exporting the descriptor array and scratchpad to user space.

4.6 StackMap Implementation

In order to validate the StackMap architecture, we implemented it in the Linux 4.2 kernel with netmap support.

```
1  // this example reuses (and omits) some of the definitions from Figure 4
2  ioctl(nmd->fd, NIOCRXSYNC);
3  uint32_t cur = ring->cur;
4  uint32_t count = 0;
5  for (uint32_t i = 0; i < nfds; i++) {
6     const int fd = fdarray[i];
7     for (uint32_t slot_idx = scratchpad[fd]; slot_idx != FD_NULL) {
8        struct netmap_slot *slot = &ring->slots[slot_idx];
9        char *buf = STACKMAP_BUF(slot->buf, ring);
10       // consume data in buf
11       slot_idx = slot->next_slot;
12       ++count;
13    }
14 }
15 // we have consumed all the data
16 cur = cur + count;
17 if (cur > ring->num_slots)
18    cur -= ring->num_slots;
19 ring->cur = ring->head = cur;
```

Figure 6: Traversing the RX ring by descriptor on receive.

To pass packets from a NIC port RX ring to the OS stack, StackMap calls `netif_receive_skb()`, and intercepts packets after this function. To pass packets from a stack port TX ring to the OS stack, StackMap calls `__tcp_push_pending_frames()` after executing some code duplicated from static functions in `tcp.c`. Packetized data is intercepted at the driver transmit routine (`ndo_start_xmit()` callback). Also, StackMap modifies the `sk_buff` destructor, TCP timer handlers and connection setup routines in Linux, and implements StackMap-specific extensions to netmap, such as `sk_buff` pre-allocation and APIs.

All in all, StackMap modifies 56 LoC (lines of code) in the netmap code, and adds 2269 LoC in a new file. In the Linux kernel, 221 LoC are added and 7 LoC are removed across 14 existing and 2 new files.

5 Experimental Evaluation

This section presents the results of a series of experiments that analyze the latency and throughput that StackMap achieves in comparison to the kernel stack used via the socket API, in order to validate our design decisions. As discussed in Section 3, StackMap focuses on improving transactional workloads with small messages and many concurrent TCP connections.

Sections 5.2 and 5.3 measure by how much StackMap reduces processing delays and queueing latencies. The experiments use a minimal HTTP server, to highlight the performance differences between StackMap and the Linux kernel.

Sections 5.4 and 5.5 measure how well StackMap performs for a realistic application (memcached), and how it competes against a Seastar [6], a highly optimized, production-quality user-space TCP/IP stack for DPDK.

5.1 Testbed Setup

The experiments use two identical Fujitsu PRIMERGY RX300 servers equipped with a 10-core Intel Xeon E5-2680 v2 CPU clocked at 2.8 GHz (3.6 GHz with Turbo Boost) and 64 GB of RAM. One machine acts as the server, the other as the client; they are connected via an Arista DCS-7050QX switch and 10 Gbit Ethernet NICs using the Intel 82599ES chipset. The multi-core experiments in Section 5.5 use two additional, similar machines connected to the same switch, to saturate the server.

The server machine runs either an unmodified Linux 4.2 kernel, our StackMap implementation (see Section 4.6) or the Seastar user-level TCP/IP stack. For all experiments involving HTTP workloads, the server executes a minimal HTTP server that uses either the socket or StackMap APIs. In the former case, the HTTP server runs an `epoll_wait()` event loop and iterates over the returned descriptors. For each returned descriptor, it (1) fetches events using `epoll_wait()`, (2) `read()`s the client request, (3) matches the first four bytes against "GET␣", (4) copies a pre-generated, static HTTP response into a response buffer and (5) `write()`s it to the descriptor. In the latter case (using the StackMap API), the HTTP server runs an `ioctl()` event loop, as described in Section 4.5 with the same application logic *i.e.*, (3) and (4). For all experiments involving memcached workloads, the server runs a memcached instance. For Linux, we use memcached [41] version 1.4.24, and for StackMap, we ported the same version of it, which required 1151 LoC of modifications.

The client always runs a standard Linux 4.2 kernel. To saturate the HTTP server, it runs the `wrk` [19] HTTP benchmark tool. `wrk` initiates a given number of TCP connections and continuously sends HTTP GETs over them, measuring the time until a corresponding HTTP OK is received. Each connection has a single outstanding HTTP GET at any given time (`wrk` does not pipeline); open connections are reused for future requests. In the experiments involving memcached, the client executes the `memaslap` [36] tool.

Except for Section 5.5, the server uses only a single CPU core to serve requests, because in these first experiments, we are interested in how well StackMap solves the problems described in Section 3. The client uses all its 10 CPU cores with receive-side-scaling (RSS) to efficiently steer traffic towards them. Unless otherwise stated, the experiments enable all hardware/software offload facilities for the experiments that use the socket API. For StackMap, such offloads are *disabled*, because netmap at the moment does not support them. Once such support is added to netmap, we expect StackMap to directly benefit from these improvements.

Configuration	64 B	512 B	1280 B	memcached
Linux	32.7 σ = 5.0	46.4 σ = 3.7	68.4 σ = 4.1	52 σ = 9.7
Linux-NIM	19.5 σ = 2.5	21.5 σ = 2.8	23.7 σ = 3.2	26 σ = 7.5
StackMap	18.6 σ = 2.8	20.6 σ = 2.9	22.7 σ = 3.2	23 σ = 5.9

Table 2: Mean roundtrip latencies in µs with standard deviations σ for different response message sizes in a granularity `wrk` or `memaslap` reports.

5.2 Processing Delay

The first experiment highlights the baseline latency of StackMap, *i.e.*, for a single message exchange without concurrency and therefore without any queueing.

Table 2 shows the mean latencies and their standard deviations for single request-response exchanges as measured by `wrk`, as well as with the `memaslap` memcached client. With `wrk`, the request message is always 96 B, and the response size varies between 64 B, 512 B and 1280 B, plus a 63 to 65 B HTTP header in each case. The memcached workload is described in Section 5.4. For the regular Linux stack, Table 2 reports two measurements. "Linux", where the NIC interrupt moderation period has been set to 1 µs (which is the Linux default), and "Linux-NIM", where interrupt moderation has been disabled. The "Linux-NIM" 64 B response size measurements were also used as the basis for the latency drill-down in Table 1 in Section 3.1.

StackMap achieves latencies that are better than Linux-NIM by 0.9 to 1 µs, which may seem minor. However, this is in fact a significant result, because both StackMap and Linux share most of the network protocol logic (Ethernet & IPv4 RX, TCP RX, TCP TX and IPv4 & Ethernet TX in Table 1) as well as the application logic (to generate "OK" replies). The StackMap latency improvement is a result of replacing the driver RX and TX operations with netmap, eliminating the socket enqueue and `epoll_wait()` operations, and replacing `read()` and `write()` with shared memory accesses by bypassing the `vfs` layer. Since these replaced or eliminated parts take 1.79 µs, this result in fact means that StackMap eliminates half of this processing overhead.

Note that for this experiment, busy-waiting on the NIC should not contribute to the latency reduction. Since Linux-NIM busy-waits on `epoll_wait()` to prevent its thread from sleeping and to avoid the thread wakeup latency, handling an interrupt (entirely done on the kernel stack of the current thread) is very cheap. We confirmed this by running netmap between the same machines, using busy-wait on either the NIC or a `epoll_wait()` descriptor. The round-trip latencies are 11.70 and 11.53 µs, respectively. This demonstrates that noticing a new packet by handling an interrupt in an active thread is slightly cheaper than through an explicit device access.

Also note that for Linux-NIM and StackMap, the latency differences between different message sizes do not result from data copies at the server or client, but are due to two 10 Gbit Ethernet and four 16 Gbit PCIe bus traversals for each of the packets, which approximately translates into 4 µs for a 1.2 KB size difference.

The "Linux" configuration is used in the rest of this section, and was used for experiments in Section 3.2. While this configuration exhibits higher latencies for a small number of concurrent connections, it achieves 28 to 78 % higher throughput and 22 to 44 % lower average latencies than Linux-NIM with 40 or more concurrent connections (not shown in the graphs).

5.3 Queueing Latency

This section evaluates by how much StackMap can reduce transaction latencies in the presence of concurrent connections (*i.e.*, transactions) compared to a system that uses the regular socket API and packet I/O methods (which were identified as inefficient in Sections 3.1 and 3.2). Recall that larger numbers of registered descriptors increase latencies because of queueing, particularly tail latencies.

The top row of graphs in Figure 7 compares the mean and 99$^{\text{th}}$-percentile latencies of StackMap and Linux. The middle row compares the mean numbers of ready descriptors returned during each event processing cycle (`ioctl()` in StackMap and `epoll_wait()` in Linux), together with their standard deviations. The bottom row shows aggregate throughputs across all connections, to validate that the improved latency does not stem from a reduction in throughput. Each column shows results for the same response size (64, 512 and 1280 B).

The results in Figure 7 match our expectations. Because StackMap reduces per-message processing delays, it can serve the same number of descriptors at a lower latency than Linux. This faster turnaround time leads to fewer descriptors that need to queue for the next round of event processing, as reflected by the lower numbers of returned ready descriptors for StackMap (middle row). As a result, StackMap increasingly outperforms Linux as the number of concurrent connections increases.

5.4 Memcached Performance

After validating that StackMap outperforms Linux for a simple application, this section evaluates how StackMap performs for a realistic application. We also compare StackMap against memcached on Seastar [6], a highly-optimized user-space TCP/IP stack for DPDK.

The experiment uses a default workload of `memaslap`, which comprises of 10 % "set" and 90 % "get" operations

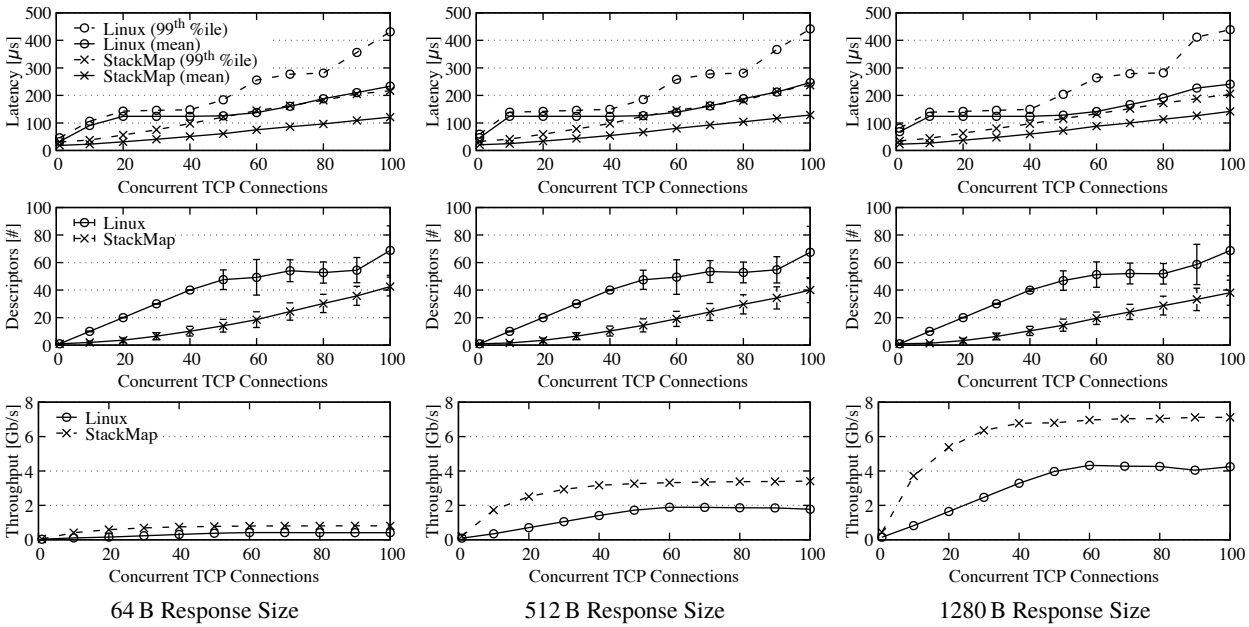


Figure 7: Mean and 99th percentile round-trip latencies (top row), mean number of ready descriptors in each event processing cycle (middle row) and throughputs (bottom row), with the number of concurrent TCP connections (horizontal axis) for different response sizes (64, 512 and 1280 B).

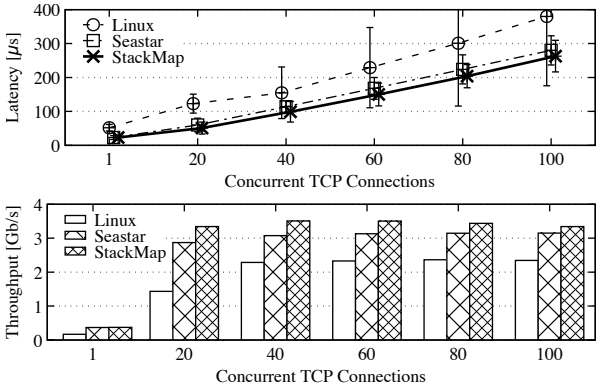


Figure 8: Memcached latency and throughput results.

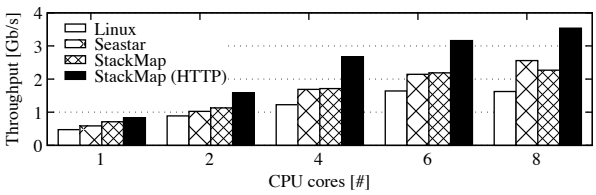


Figure 9: Memcached multi-core throughput.

on 1024 B objects. While still simple, memcached has a slightly more complex application logic than the simple HTTP server we used for the previous benchmarks, and therefore exhibits higher processing delays (see Table 2).

Figure 8 shows mean latencies with standard deviations, as well as aggregate throughputs. StackMap achieves significantly higher throughputs and lower latencies than Linux, as well as a much smaller latency variance. This is similar to observations earlier in this section. Surprisingly, StackMap also slightly outperforms Seastar.

5.5 Memcached Multicore Scalability

Finally, we evaluate multi-core scalability with StackMap, again using memcached as an application, and compares the results against Linux and Seastar. The object size for this experiment is 64 B to prevent the network from becoming saturated. In order to investigate the overheads and scalability of memcached itself, we also compare against the simple HTTP server used for our other measurements, configured to also serve 64 B messages.

Figure 9 shows aggregate throughputs when using a different number of CPU cores to serve the workload. Up to six cores used, the relative performance differences between Linux, Seastar and StackMap remain similar, with Linux being slowest, and StackMap slightly outperforming Seastar. However, Seastar begins to outperform the others at eight cores. StackMap and Linux retain their relative performance difference. This is to be expected, because the current StackMap implementation does not yet optimize locking when the Linux TCP/IP operates in StackMap mode, nor has memcached been modified to remove such locks when used with StackMap. In contrast, Seastar adopts a shared-nothing design, and memcached on top of Seastar also has been highly optimized for multi-core scalability. The inclusion of the simple HTTP server results for StackMap attempt to illustrate its scal-

ability potential (and also illustrate the need for future improvements to StackMap).

6 Limitations and Future Work

This section briefly discusses limitations of the StackMap architecture and prototype implementation, as well as potential future improvements.

6.1 Misbehaving Applications

The OS packet representation structure (`sk_buff` or `mbuf`) consists of metadata and a pointer to a data buffer, whereas StackMap exposes only data buffers to its applications. However, Linux also stores some metadata at the end of a data buffer (`skb_shared_info`), which includes information to share a data buffer between multiple `sk_buff`s and pointers to additional data buffers [9].

Thus, system memory could leak—or the system may crash—if a misbehaving StackMap application modifies this metadata. One possible solution is to link a netmap buffer as an additional buffer, because Linux stores no metadata for those, and to only use a primary data buffer to store `skb_shared_info`. A similar method would also work for FreeBSD, using external storage associated with an `mbuf`. We will explore such solutions in the future.

StackMap also increases the risk of the network stack malfunction when a StackMap application misuses the netmap API or it modifies buffers owned by the kernel (indicated by the head and tail pointers [53]). In the original netmap, kernel-owned buffers are only touched by the netmap backend, which is robust against the case where an application modifies data out of turn. However, with StackMap, these kernel-owned buffers are also processed and referred to by the kernel TCP/IP stack. Therefore, if such buffers are modified by the application out of turn (*e.g.*, modifying the sequence number of a sent packet that is referred to from the retransmission queue), the TCP/IP implementation could fall into inconsistent state.

Possible solutions to this issue include providing a wrapper API to prevent the application from accessing the kernel-owned buffers, making the kernel create a private copy of data and making the kernel its buffers read-only. Since any of these mitigation methods comes at some cost, we leave their investigation as future work.

6.2 Support for Other Protocols

Although the StackMap prototype implementation supports only TCP and IPv4, the StackMap architecture is not limited to these protocols. However, UDP-based applications can already batch syscalls using `sendmmsg()` and `recvmmsg()`, may not need an event multiplexing API like `epoll_wait()` and can exploit batching during packet I/O for transmission of messages to different clients [30]. Therefore, the performance benefits that the StackMap architecture could bring to new UDP-based protocols such as QUIC [20] need to be investigated.

6.3 System Configuration

In order to achieve the best network performance, system administrators should configure their systems such that traffic to and from regular applications are routed via NICs that are not used by any StackMap application. Not doing so does not crash the system, but regular applications could see unexpected packet delays, because moving packets in and out of the NICs is triggered by the StackMap application, and not the normal kernel methods. Nevertheless, in many of today's production systems, such configuration is already regularly performed, and so does not complicate StackMap deployment.

7 Conclusion

Our goal in this paper has been to address the latency problems of transactional workloads over TCP, which consist of small messages sent over a large number of concurrent connections. We demonstrated that the kernel TCP/IP implementation is reasonably fast, but showed that the socket API and the traditional packet I/O methods increase transaction latencies and limit throughputs. StackMap, a new interface to the OS TCP/IP service that exploits the opportunities afforded by dedicating NICs to applications, addresses these performance issues. The StackMap design challenges included combining the full-featured TCP/IP implementation in the kernel with netmap, which in addition to fast packet I/O methods provides protection of the system and NICs, and a sophisticated API.

A key takeaway is that an integration of most of the techniques introduced by high-performance kernel-bypass TCP/IPs—including new APIs, syscall and I/O batching, lightweight buffer management and direct packet buffer access—can be leveraged smoothly into the OS stack, and help it achieve comparable performance. The key advantage of this approach is that StackMap allows applications to enjoy modern TCP/IP features and benefit from the active maintenance that the OS stack is seeing, which kernel-bypass TCP/IPs lack.

8 Acknowledgments

This paper has received funding from the European Union's Horizon 2020 research and innovation program 2014–2018 under grant agreement No. 644866 ("SSICLOPS"). It reflects only the authors' views and the European Commission is not responsible for any use that may be made of the information it contains.

References

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. "Data Center TCP (DCTCP)". *Proc. ACM SIGCOMM*. 2010.

[2] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. "pFabric: Minimal Near-optimal Datacenter Transport". *Proc. ACM SIGCOMM*. 2013.

[3] M. Allman, H. Balakrishnan, and S. Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC 3042. Jan. 2001.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency". *Proc. USENIX OSDI*. 2014.

[5] A. Bittau, M. Hamburg, M. Handley, D. Mazières, and D. Boneh. "The Case for Ubiquitous Transport-level Encryption". *Proc. USENIX Security*. 2010.

[6] Cloudius Systems. *Seastar*. http://www.seastar-project.org/.

[7] J. Corbet. *The kernel connection multiplexer*. https://lwn.net/Articles/657999/. Sep. 21, 2015.

[8] J. Cummings and E. Tamir. *Open Source Kernel Enhancements for Low Latency Sockets using Busy Poll*. Intel White Paper. 2013.

[9] David S. Miller. *How SKBs work*. http://vger.kernel.org/~davem/skb_data.html.

[10] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01. Feb. 25, 2013.

[11] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. "Proportional Rate Reduction for TCP". *Proc. ACM IMC*. 2011.

[12] A. Dunkels. "Design and Implementation of the lwIP TCP/IP Stack". *Swedish Institute of Computer Science*, 2001.

[13] W. Eddy. *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987. Aug. 2007.

[14] F. C. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston, and B. Chen. *Architecture of Systemtap: A Linux Trace/Probe Tool*. 2005.

[15] M. Eisler, P. Corbett, M. Kazar, D. S. Nydick, and C. Wagner. "Data ONTAP GX: A Scalable Storage Cluster". *Proc. USENIX FAST*. 2007.

[16] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. "Lazy Asynchronous I/O for Event-driven Servers". *Proc. USENIX ATC*. 2004.

[17] M. Al-Fares, A. Loukissas, and A. Vahdat. "A Scalable, Commodity Data Center Network Architecture". *Proc. ACM SIGCOMM*. 2008.

[18] A. Ford, C. Raiciu, M. J. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824. Oct. 14, 2015.

[19] GitHub. *Modern HTTP benchmarking tool*. https://github.com/wg/wrk. Jul. 2013.

[20] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-tsvwg-quic-protocol-02. Jan. 13, 2016.

[21] S. Han, K. Jang, K. Park, and S. Moon. "PacketShader: A GPU-accelerated Software Router". *Proc. ACM SIGCOMM*. 2010.

[22] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. "MegaPipe: A New Programming Interface for Scalable Network I/O". *Proc. USENIX OSDI*. 2012.

[23] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. "mSwitch: A Highly-scalable, Modular Software Switch". *Proc. ACM SOSR*. 2015.

[24] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. "Rekindling Network Protocol Innovation with User-level Stacks". *ACM SIGCOMM CCR*, Apr. 2014.

[25] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. "Is It Still Possible to Extend TCP?": *Proc. ACM IMC*. 2011.

[26] Intel. *Intel DPDK: Data Plane Development Kit*. http://dpdk.org/. Sep. 2013.

[27] Intel. *Introduction to the Storage Performance Development Kit (SPDK)*. https://software.intel.com/en-us/articles/introduction-to-the-storage-erformance-development-kit-spdk. Sep. 18, 2015.

[28] V. Jacobson, R. Braden, and D. Borman. *TCP Extensions for High Performance*. RFC 1323. May 1992.

[29] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems". *Proc. USENIX NSDI*. 2014.

[30] Jesper Dangaard Brouer. *Unlocked 10Gbps TX wirespeed smallest packet single core*. http://netoptimizer.blogspot.de/2014/10/unlocked-10gbps-tx-wirespeed-smallest.html.

[31] A. Kantee. "Environmental Independence: BSD Kernel TCP/IP in Userspace". *AsiaBSDCon*, 2009.

[32] M. Larsen and F. Gont. *Recommendations for Transport-Protocol Port Randomization*. RFC 6056. Jan. 2011.

[33] J. Leverich and C. Kozyrakis. “Reconciling High Server Utilization and Sub-millisecond Quality-of-service”. *Proc. ACM EuroSys*. 2014.

[34] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. “The Emergence of Networking Abstractions and Techniques in TinyOS”. *Proc. USENIX NSDI*. 2004.

[35] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. “Scalable Kernel TCP Design and Implementation for Short-Lived Connections”. *Proc. ACM ASPLOS*. 2016.

[36] M. Zhuang and B. Aker. *memaslap: Load testing and benchmarking a server*. http://docs.libmemcached.org/bin/memaslap.html.

[37] I. Marinos, R. N. Watson, and M. Handley. “Network Stack Specialization for Performance”. *Proc. ACM SIGCOMM*. 2014.

[38] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018. Oct. 1996.

[39] M. Mathis and J. Mahdavi. “Forward Acknowledgement: Refining TCP Congestion Control”. *Proc. ACM SIGCOMM*. 1996.

[40] A. Medina, M. Allman, and S. Floyd. “Measuring the Evolution of Transport Protocols in the Internet”. *ACM SIGCOMM CCR*, Apr. 2005.

[41] *memcached - a distributed memory object caching system*. http://memcached.org/.

[42] Microsoft. *Windows I/O Completion Ports*. Microsoft White Paper. 2012.

[43] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. “The Click Modular Router”. *Proc. ACM SOSP*. 1999.

[44] V. Paxson, M. Allman, S. Dawson, W. Fenner, J. Griner, I. Heavens, K. Lahey, J. Semke, and B. Volz. *Known TCP Implementation Problems*. RFC 2525. Mar. 1999.

[45] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. “Fastpass: A Centralized "Zero-queue" Datacenter Network”. *Proc. ACM SIGCOMM*. 2014.

[46] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. “Improving Network Connection Locality on Multicore Systems”. *Proc. ACM EuroSys*. 2012.

[47] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. “Arrakis: The Operating System is the Control Plane”. *Proc. USENIX OSDI*. Oct. 2014.

[48] J. Postel. *Transmission Control Protocol*. RFC 793. Sep. 1981.

[49] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. “TCP Fast Open”. *Proc. ACM CoNEXT*. 2011.

[50] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. “How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP”. *Proc. USENIX NSDI*. 2012.

[51] A. Ramaiah, R. Stewart, and M. Dalal. *Improving TCP’s Robustness to Blind In-Window Attacks*. RFC 5961. Aug. 2010.

[52] L. Rizzo, M. Carbone, and G. Catalli. “Transparent Acceleration of Software Packet Forwarding using netmap”. *Proc. IEEE Infocom*. Mar. 2012.

[53] L. Rizzo, G. Lettieri, and M. Honda. “Netmap as a Core Networking Technology”. *AsiaBSDCon*, 2014.

[54] L. Rizzo. “netmap: A Novel Framework for Fast Packet I/O”. *Proc. USENIX ATC*. Jun. 2012.

[55] L. Rizzo. “Revisiting Network I/O APIs: The Netmap Framework”. *Queue*, Jan. 2012.

[56] L. Rizzo and G. Lettieri. “VALE, a Switched Ethernet for Virtual Machines”. *Proc. ACM CoNEXT*. 2012.

[57] P. Sarolahti and M. Kojo. *Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP)*. RFC 4138. Aug. 2005.

[58] P. Sarolahti, M. Kojo, and K. Raatikainen. “F-RTO: An Enhanced Recovery Algorithm for TCP Retransmission Timeouts”. *ACM SIGCOMM CCR*, Apr. 2003.

[59] L. Soares and M. Stumm. “FlexSC: Flexible System Call Scheduling with Exception-less System Calls”. *Proc. USENIX OSDI*. 2010.

[60] H. Tazaki, R. Nakamura, and Y. Sekiya. “Library Operating System with Mainline Linux Network Stack”. *Proc. netdev0.1*. Feb. 2015.

[61] The Open Group. *Networking Services, Issue 4*. Sep. 1994.

[62] M. Zec, L. Rizzo, and M. Mikuc. “DXR: Towards a Billion Routing Lookups Per Second in Software”. *ACM SIGCOMM CCR*, Sep. 2012.

SLIK: Scalable Low-Latency Indexes for a Key-Value Store

Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang and John Ousterhout

Stanford University

Abstract

Many large-scale key-value storage systems sacrifice features like secondary indexing and/or consistency in favor of scalability or performance. This limits the ease and efficiency of application development on such systems. Implementing secondary indexing in a large-scale memory based system is challenging because the goals for low latency, high scalability, consistency and high availability often conflict with each other. This paper shows how a large-scale key-value storage system can be extended to provide secondary indexes while meeting those goals. The architecture, called SLIK, enables multiple secondary indexes for each table. SLIK represents index B+ trees using objects in the underlying key-value store. It allows indexes to be partitioned and distributed independently of the data in tables while providing reasonable consistency guarantees using a lightweight ordered write approach. Our implementation of this design on RAMCloud (a main memory key-value store) performs indexed reads in 11 μs and writes in 30 μs. The architecture supports indexes spanning thousands of nodes, and provides linear scalability for throughput.

1 Introduction

Over the last decade, a new class of storage systems has arisen to meet the needs of large-scale web applications. Various main-memory-based data storage systems such as Aerospike [1], H-Store [19], RAMCloud [24] and Redis [8] have scaled to span hundreds or thousands of servers, with unprecedented overall performance. However, in order to achieve their scalability, most large-scale storage systems have accepted compromises in their feature sets and consistency models. In particular, many of these systems are simple key-value stores with no secondary indexes. The lack of secondary indexes makes it difficult to implement applications that need to make range queries and/or retrieve data by keys other than the primary key.

Indexing has been studied extensively in the context of traditional databases. However, its design for a low-latency large-scale main-memory storage system presents several unique design issues (given below). These are further challenging due to the inherent tension between some of them.

- Low latency: The system should harness low latency networks, store index data in DRAM, and leave out complex mechanisms wherever possible in favor of lightweight methods that add minimal overhead.
- Scalability: A large-scale data store must support tables so large that their objects and indexes need to span many servers. The total throughput of an index should increase linearly with the number of servers it spans. This objective is at odds with low latency, as contacting more servers (even if done in parallel) increases latency. Ideally, a system should provide nearly constant latency irrespective of the number of servers an index spans.
- Consistency: The system should provide clients with the same strong consistency as a centralized system. For instance, when an indexed object is written, the update to that object and all of its indexes must appear atomic, even in the face of concurrent accesses and server crashes. However, providing consistency when information is distributed, traditionally requires locks or algorithms that impact latency or scalability. Further, as data and indexes become sharded over more and more nodes, it becomes increasingly complex and expensive to manage metadata and maintain consistency between data and the corresponding indexes.
- Availability: The system must also be continuously available; this creates challenges around crash recovery and requires that schema changes such as adding and removing indexes be accomplished without taking the system offline.

In this paper, we show how to overcome these challenges and how a large-scale key-value store can be extended to provide secondary indexes. The resulting architecture, SLIK (Scalable, Low-latency Indexes for a Key-value store), combines several attractive features. First, it scales to provide high performance even with indexes that span hundreds of servers while providing strong consistency guarantees. Second, its mechanisms are simple enough to provide extraordinarily low latency when used with a low-latency key-value store. Third, it provides fast crash recovery, live index split and migration and other features that ensure a high level of availability. Finally, it uses main memory judiciously

while storing secondary index structures.

SLIK uses several interesting approaches to achieve the desired properties:

- Its data model is a multi-key-value store, where each object can have multiple secondary keys in addition to the primary key and an uninterpreted data blob. This approach reduces parsing overheads for both clients and servers to improve latency.
- SLIK achieves high scalability by distributing index entries independently from their objects rather than colocating them (which is the more commonly used approach today).
- However, the resulting indexed operations are now distributed, which creates potential consistency problems between indexes and objects. SLIK provides clients with a consistent behavior using a novel lightweight mechanism that avoids the complexity and overhead of distributed transactions. It uses an ordered-write approach for updating indexed objects and uses objects as ground truth to determine liveness of index entries.
- SLIK performs long-running bulk operations such as index creation/deletion and migration in the background, without blocking normal operations. For example, SLIK uses a logging approach for index migration, which allows updates to an index as it is being migrated.
- Finally, SLIK implements secondary indexes using an efficient B+ Tree algorithm. Each tree node is kept compact by mapping secondary keys to the primary key hashes of the corresponding objects. SLIK further uses objects of the underlying key-value store to represent these nodes, and leverages the existing recovery mechanisms of the key-value store to recover indexes.

To demonstrate the practicality of SLIK, we implemented it in RAMCloud [24], a low-latency distributed key-value store. The resulting system provides extremely low latency indexing and is highly scalable:

- SLIK performs index lookups in 11–13 µs, which is only 2x the base latency of non-indexed reads.
- SLIK performs durable updates of indexed objects in 30–36 µs, which is also about 2x the base latency of non-indexed durable updates.
- The throughput of an index increases linearly as it is partitioned among more and more servers.
- SLIK's latency is 5–90x lower than H-Store, a state-of-the-art in-memory database.

Overall, SLIK demonstrates that large-scale storage systems need not forgo the benefits of secondary indexes.

2 The SLIK Design

In this section we describe the general architecture of SLIK, which could be used with any underlying key-value store. In the next section we will discuss features that are specific to our implementation in RAMCloud.

2.1 Data Model

In order to have secondary indexes, clients and servers must agree on where the secondary keys are located in the object. A traditional key-value store does not provide this information, as each object only contains a single key and a value. One commonly used approach is to store the keys as part of the object's value. In this case, the servers and clients must agree on a specific format for object values, such as JSON. Here, each index is associated with a particular named field, and the server parses the object value to find the secondary keys. Several storage systems use this approach, including CouchDB [2] and MongoDB [5]. However, this approach introduces additional overhead for the server to parse objects.

Given our objective of lowest possible latency in SLIK, we chose an object structure that directly identifies all the secondary keys. Consequently, there is no parsing required to carry out index operations. We call this a *multi-key-value* format: an object consists of one or more variable-length keys, plus a variable-length uninterpreted value blob. The first key is the primary key: along with the table identifier, this uniquely identifies an object. The rest of the keys are for secondary indexes: these need not be unique within the table. Each of the secondary keys can have an index corresponding to it. Each key can be of a different type with a corresponding ordering function (for example, numerical or lexicographic).

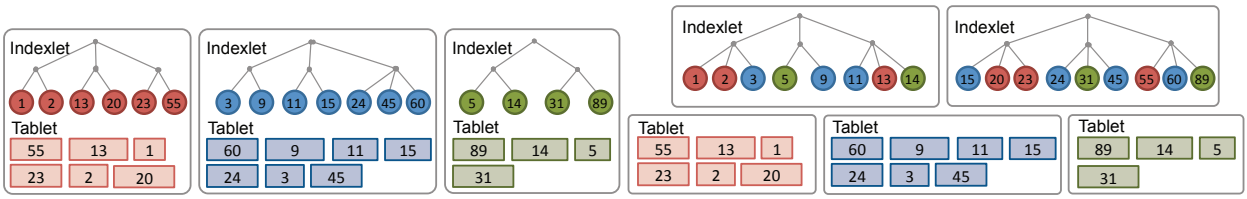
The object format can be managed automatically by client side libraries, so that applications do not have to be aware of how the information is stored in object values and secondary keys.

2.2 Index Partitioning

To be usable in any large-scale storage system, a secondary indexing system must support tables so large that neither their objects nor their indexes fit on a single server. In an extreme case, an application might have a single table whose data and indexes span thousands of servers. Thus, it must be possible to split indexes into multiple index partitions, or *indexlets*, each of which can be stored on a different server.

A scalable index performs well even if it spans many servers. The index should provide nearly constant and low latency irrespective of the number of servers it spans. Additionally, the total throughput of an index should increase linearly with the number of partitions. To design an indexing architecture that achieves these goals, we considered three alternative approaches to index partitioning.

One approach is to colocate index entries and objects, so that all of the indexing information for a given object



(a) Indexlet/Tablet Colocation (b) Independent Partitioning

Figure 1: Two approaches to index partitioning assuming that a table is partitioned by its primary key. In (a) indexes for the table are partitioned so that the index entry for each object is on the same server as the object. In (b) indexes for the table are partitioned so that each indexlet contains all the keys in a particular range. Rectangles represent objects, and the number in each rectangle is the value of the secondary key for that object (primary keys and object values are omitted). Circles represent index nodes; the number in each circle is the value of the secondary key. Colors distinguish objects (and secondary index keys) that belong to different tablets.

is stored on the same server as that object. In this approach, one of the keys (either the primary key or a specified secondary key) is used to partition the table's objects among servers, as shown in Figure 1(a). We call this ***Colocation Approach***, in which each server stores a table partition (or tablet) plus one indexlet for each of that table's indexes. The indexlet stores only index entries for the tablet on the same server. This approach is used widely by many modern storage systems, including Cassandra [20] and H-Store [19], and the local indexes in Espresso [26] and Megastore [11].

To perform a lookup using an index, a client issues parallel Remote Procedure Calls (RPCs) to all the servers holding partitions for this table. Each server scans its local indexlet, then returns the matching objects from its local tablet.

A second approach is to partition each index and table independently, so that index entries are not necessarily located on the same servers as the corresponding objects. This allows each index to be partitioned according to the sort order for that index, as shown in Figure 1(b). We call this ***Independent Partitioning***. With this approach, a small index range query can be processed entirely by a single index server.

With independent partitioning, a client performs index lookups in two steps. In the first step, the client issues an RPC to the server holding the relevant indexlet. This can typically be processed entirely by a single index server. If the queried range spans multiple indexlets, then each of the corresponding servers is contacted. This RPC returns information to identify the matching objects. The client then contacts the relevant data servers to retrieve these objects.

At small scale, the colocation approach provides lower latency. For example, in the limit of a single server, it requires only a single RPC, whereas independent partitioning requires two RPCs. However, as the number of servers increases, the performance of the colocation approach degrades. Each request must contact more and more servers, so the lookup cost increases linearly with the number of servers across which a table is sharded. On the other hand, independent partitioning provides dramatically better performance. Executing two sequential RPCs results in a constant latency (even as the number of partitions is increased), and this latency is lower than executing a large number of parallel RPCs. Moreover, with independent partitioning, the total lookup throughput increases with the addition of servers. This is not the case with the colocation approach, as each server must be involved in every index lookup. While many modern datastores use the colocation approach, our experiments in Section 4 show that the independent partitioning scheme provides better scalability.

A third approach is to use independent partitioning, but also replicate part or all of the table's data with each index. Any data that may be accessed via the index needs to be duplicated in this index. This approach is used by the global indexes in DynamoDB [3] and Phoenix [7] on HBase [4].

Global indexes combine some important benefits of the two approaches above. They enable low latency lookups as a lookup requires only a single RPC for small range queries. They are also scalable as the indexes are partitioned independently of the data table.

These benefits are at the cost of increased memory footprint: an index lookup can return only those attributes of the object that have been duplicated and stored with that index. This results in substantial data duplication, which might be acceptable for disk-based systems, but not for a memory-based system like SLIK.

We use the independent partitioning approach in SLIK. This enables high scalability while using memory efficiently.

2.3 Consistency during normal operations

As discussed in the previous section, indexed object writes and index lookups are distributed operations because objects and corresponding index entries may be stored on different servers. This creates potential consistency problems between the indexes and objects.

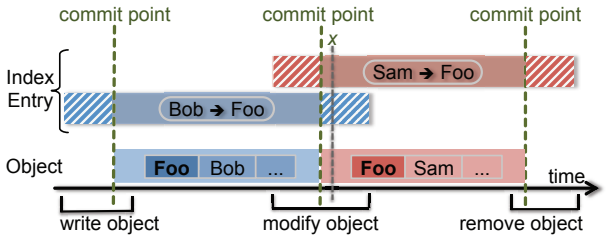


Figure 2: The ordered write approach ensures that if an object exists, then the corresponding index entries exist. Each object provides the ground truth to determine the liveness of its index entires. Writing an object serves as the commit point. The box at the bottom shows an object as it is created, modified and removed (Foo is the objects primary key; the secondary key is changed from Bob to Foo when the object is modified). The boxes above show corresponding index entries, where the solid portion indicates a live entry. At point *x*, there are two index entries pointing to the object, but the stale entry (for Bob) will be filtered out during lookups.

Existing storage systems have generally dealt with index consistency in two ways. Many large scale storage systems simply permit inconsistencies, in order to simplify their implementations or improve performance. For example, CouchDB [2], PNUTS [13], the global indexes for Espresso [26] and Megastore [11], and Tao [12] use relaxed consistency. This forces application programmers to build their own mechanisms to ensure consistency. The second approach, typical of smaller-scale systems, is to wrap updates in transactions that ensure atomicity. However, we were concerned about the implementation complexity and potential scalability problems of using transactions for this purpose.

Our goal is to build a scalable distributed system with the consistency expected from a centralized system. SLIK provides clients with the same behavior as if indexes and objects were on the same server with locks to control access. More concretely, SLIK guarantees the following consistency properties:

1. If an object contains a given secondary key, then an index lookup with that key will return the object.
2. If an object is returned by an index lookup, then this object contains a secondary key for that index within the specified range.

We want to provide this consistency while imposing miminal performance overheads. We designed a simple lighweight mechanism that ensures the consistency properties stated above without requiring transactions. It guarantees the first property by using an ordered write approach. It guarantees the second property by treating index entries as hints and using objects as ground truth to determine the liveness of index entries. This mechanism is explained in detail below, and illustrated in Figure 2.

SLIK uses an *ordered write approach* to ensure that the lifespan of each index entry spans that of the corresponding object. Specifically, when a data server receives a write request, it first issues requests (to the server(s) with relevant indexlets) for creating index entries corresponding to each of the secondary keys in the new object's data. Then it writes the object and replicates it durably. Finally, it asynchronously issues requests (again, to the server(s) with relevant indexlets) for removing old index entries, if this was an overwrite. This means that if an object exists, then the index entries corresponding to it are guaranteed to exist – thus ensuring the first of the two consistency properties.

However, now it is possible for a client to find index entries that refer to objects that have not yet been written or no longer exist – this would violate the second consistency property. To solve this, we observe that the information in an object is the *truth* and index entries pointing to it can be viewed as *hints*. During index range queries, the client first queries the indexlet server(s) responsible for the requested range. These servers identify the matching objects by returning a hash of the primary key for each matching object (Section 2.6 discusses the use of primary key hashes in detail). The client then uses these primary key hashes to fetch all of the corresponding objects. Some of these objects may not exist, or they may be inconsistent with the index (see Figure 2, point *x*). The SLIK client library detects these inappropriate objects by rechecking the actual index key present in each object. Only objects with keys in the desired range are returned to the application.

Writing an object effectively serves as a *commit point* – any index entries corresponding to the current data are now live, and any old entries pointing to it are now dead. This ensures the second of the two consistency properties.

The SLIK approach permits temporary inconsistencies in internal data structures but masks them to provide the client applications with a consistent view of data. This results in a relatively simple and efficient implementation, while giving client applications the consistent behavior defined by the two properties above.

2.4 Metadata and Coordination

The metadata about the mapping from indexlets to their host servers needs to be managed using a persistent coordination service. This metadata is updated when a new index is created or dropped, an index server crashes or recovers data from another crashed server, and when an indexlet is split or migrated to another node. The co-ordination service only stores and disseminates the metadata: it does not take part in any lookup or write operations.

2.5 SLIK API

Tables 1 and 2 summarize the API of SLIK: Table 1 shows the operations visible to client applications, and

Table 2 shows the internal RPCs used to implement them.

A client invokes `createIndex` and `dropIndex` to create or delete an index on an existing data table (identified by `tableId`). The index is identified by `indexId`, such that the *n*th key in the object is indexed by index with `indexId` *n*.

When a client starts an index lookup, the SLIK API library acts as overall manager. It first issues `lookupKeys` to the appropriate index servers. Each index server identifies the primary key hashes for the objects in the secondary key range and returns them in index order. Then the client issues `readHashes` in parallel to the relevant data servers to fetch the actual objects using the primary key hashes. The objects returned from each data server are also in index order (as the order was specified by the key hashes in the query). For large range queries, SLIK uses a concurrent and pipelined approach with multiple RPCs in flight simultaneously. It is implemented using a rules-based approach [29]. As it receives the responses from various servers, it prunes extraneous entries (as per the consistency algorithm in Section 2.3) and collates results from different RPCs, so that the objects are returned to the client in index order.

We used a streaming approach (with an iterator API) rather than an approach that collects and returns all the objects at once. This allows index range queries to return very large result sets, which might not all fit in client memory at once.

To write an indexed object, a client sends a `write` request to the data server that stores the object. The data server synchronously issues `entryInsert` requests to relevant index servers to add new index entries, then modifies the object locally and durably replicates it. At this point, the data server returns a response to the client, then asynchronously issues `entryRemove` requests to relevant index servers. If the object is new (it did not previously exist), then the index removal step is skipped.

2.6 Index Storage and Durability

SLIK uses a B+ tree to represent each indexlet, so that range queries can be supported. The B+ tree nodes map secondary keys to the corresponding objects. However, as SLIK uses the independent partitioning approach, index entries need a way to identify the objects they refer to. A straightforward way is for an index entry to map its secondary key to the primary key of the corresponding object. However, primary keys are variable length byte arrays, which can potentially be large (many KBs), so SLIK indexes identify an object instead with a 64-bit hash value computed from its primary key. Primary key hashes have the advantage of being shorter and fixed in size. A compressed form of the key, such as a hash, works just as well as using the entire key, as it finds the right server and does not miss any objects.

`createIndex(tableId, indexId, indexType)` Create a new index for an existing table.
`dropIndex(tableId, indexId)` Delete the specified index. Secondary keys in existing objects are not affected.
`IndexLookup(tableId, indexId, firstKey, lastKey, flags)` Initiate the process of fetching objects whose keys (for index `indexId`) fall in the given range. `flags` provide additional parameters (for example, whether the end points of the range should be included in the search). This constructs an iterator object.
`IndexLookup::getNext()` Get the next object in index order as per parameters specified earlier in `IndexLookup`, or wait until such an object is available.
`write(tableId, keys, value)` Create or overwrite the object. Update secondary indexes both to insert new secondary keys and to remove old ones (if this was an overwrite).

Table 1: A summary of the core API provided by SLIK to client applications for managing indexes and secondary keys.

`lookupKeys(tableId, indexId, firstKey, lastKey, flags)` Returns primary key hashes for all entries in the given index in the given range. `flags` provide additional parameters (for example, whether the end points of the range should be included in the search).
`readHashes(tableId, pKHashes)` Returns objects in table (`tableId`) whose primary key hash matches one of the hashes in `pKHashes`.
`entryInsert(tableId, indexId, key, pKHash)` Adds a new entry to the given index. This entry maps the secondary key (`key`) to a primary key hash (`pKHash`). Replicates the update durably before returning.
`entryRemove(tableId, indexId, key, pKHash)` Removes the given entry in the given index. Replicates the update durably before returning.

Table 2: A summary of the core RPCs used internally by SLIK to implement the `IndexLookup` and `write` operations in Table 1. Additional operations for managing indexlet ownership are omitted here.

It may occasionally select extra objects, but these extra objects get pruned out as a by-product of the consistency algorithm.

SLIK keeps these B+ trees entirely in DRAM in order to provide the lowest possible latency. However, index information must be as durable and available as the objects in the key-value store (for example, it must survive server crashes).

One approach for achieving index durability is to rebuild indexlets from table data. To recover an indexlet with the ***rebuild approach***, each server storing objects of the corresponding table reads all the objects in its memory to find keys that belong to the crashed indexlet. Then the server that is the new owner of this indexlet reconstructs the indexlet using the table data. This approach is attractive for two reasons. First, it is simple: indexlets can be managed without worrying about durability. Second, it offers high performance: there is no need to replicate index entries or copy them

to nonvolatile storage such as disk or flash. However, the rebuild approach does not allow fast crash recovery: our tests show that it will take 25 seconds or more to recover a 500 MB partition, and the time (for the same sized partition) will increase as server memory sizes increase.

Our goal for crash recovery is to recover lost index data in about the same amount of time that the underlying storage system needs to recover lost table data. To achieve this, SLIK represents each indexlet B+ tree with a *backing table* in the underlying key-value store; the backing table is just like any other table, except that it is not visible to clients and has a single tablet. Each node in the B+ tree is represented with one object in the backing table. This ***backup approach*** allows indexlets to leverage the persistence and replication mechanisms the underlying key-value store uses for its object data.

With this approach, index crash recovery consists primarily of recovering the corresponding backing table. This is handled by the underlying key-value store. Once the backing table becomes available, the indexlet is fully functional; there is no need to reconstruct a B+ tree or to scan objects to extract index keys. Thus, this approach allows indexes to be recovered just as quickly as objects in the underlying key-value store.

The backup approach to index recovery does have two disadvantages. First, since each node in the B+ tree is a separate object in the key-value store, traversing a pointer from a node to one of its children requires a lookup in the key-value store (pointers between nodes are represented as keys in the key-value store). This is slightly more expensive than dereferencing a virtual memory address, which would be the case if the B+ tree nodes were not stored using objects. Second, the backup approach requires an object to be written durably during each index update, whereas the rebuild approach would not require this step. This durable write affects the performance of index updates (as shown by the measurements in Section 4).

However, the backup approach has another major advantage of permitting variable-size nodes in index B+ trees. Many B+ tree implementations (such as MySQL/InnoDB [6]) allocate fixed size B+ tree nodes. This results in internal fragmentation when the index keys are of variable length (as with commonly used strings). Since key-value stores naturally permit variable-size objects, the nodes in SLIK's B+ trees can also be of variable size, which eliminates internal fragmentation and simplifies allocation.

2.7 Consistency after Crashes

SLIK must handle additional consistency issues that may arise due to server or client crashes.

2.7.1 Server Crash

A server crash can create two consistency issues. First, if a server crashes after inserting an index entry but before updating the object (or after updating an object but before removing old index entries), the crash may leave behind extraneous index entries that will not be deleted by normal operations.

These entries can be garbage collected by a background process. Occassionally, this process scans the indexes and sends the entries to relevant data servers. For each index entry, the data server acquires a lock that prevents concurrent accesses to the corresponding object. It then checks whether the object exists. If the object does not exist, the data server sends an `entryRemove` request to the index server. If the table partition corresponding to an entry is being recovered, the collector simply skips that entry: it will be removed during the next scan.

We chose to exclude this garbage collector from our implementation as it would have added complexity for little benefit. The orphan entries do not affect correctness as they get filtered out during lookups by the consistency algorithm in the previous section. Further, the wasted space from these entries would be negligible. Assuming conservatively a mean time to failure for servers of about 4 months [18], 10 indexed object writes (or overwrites) in progress at the time of a crash, and 100 B for each index entry, the total amount of garbage accumulated will be less than 3 KB per server per year.

The second consistency issues arises if an indexlet server crashes while inserting or removing an entry, it can cause consistency issues in the internal B+ tree structure. Index insertions and deletions may cause nodes of the B+ tree to be split or joined, which requires updates to multiple nodes (and the objects that encapsulate each node). In order to maintain the consistency of the index across server crashes, multi-object updates must occur atomically. SLIK uses a multi-object update mechanism implemented using the log-structured memory or transaction log of the underlying key-value store. This ensures that after a crash, either all or none of the updates will be visible.

2.7.2 Client Crash

SLIK has been designed such that a client crash does not affect consistency: all operations that have consistency issues (like write) are managed by servers. Consequently, a client crash does not require any recovery actions other than closing network connections.

2.8 Large Scale Operations vs. Scalability

To maximize scalability, large-scale long-running operations must not block other operations. The number of other operations blocked by a given operation is likely to be proportional to the size of the data set blocked.

This means that an operation may hold a lock on a small amount of data for a comparatively long amount of time while a lock on a larger set of data needs to be held for a short amount of time. Hence, SLIK performs long-running bulk operations such as index creation/deletion and migration in the background, without blocking normal operations.

2.8.1 Index Creation

When a new index is created for an existing table in SLIK, it needs to be populated with the index key information from the table's objects. This requires a scan of the entire table, which could take a considerable amount of time for a large table. Furthermore, objects in the table may need to be reformatted to include the corresponding secondary keys, which requires rewriting the objects in the table.

One approach is to lock the table for the duration of table scanning and object rewriting. However, this is not scalable: as tables get larger and larger, the lock must be held for a longer and longer time period, during which period normal requests cannot be serviced.

In order to allow the system to function normally even during schema changes, SLIK populates a new index in the background, without locking the table. The new index should not be used for lookups until index creation is complete. However, the table is locked only long enough to create an empty indexlet. Once the lock is released, other operations on the table can be serviced while the index is being populated. For example, lookups on other indexes (or the primary key) can be serviced. Additionally, objects can be written into the table. These writes will update the new index as well as existing ones.

To populate the new index with entries corresponding to the objects already in the underlying table, client-level code scans this table, reading each object and then rewriting it. Before rewriting the object, the client can restructure the object if the schema has changed. The act of rewriting the object creates an entry in the new index corresponding to this object. So, once all of the objects have been scanned and rewritten, the index is complete. The index population operation is idempotent; if it is interrupted by a crash, it can be restarted from the beginning.

Index deletion behaves similarly to index creation. The index can be deleted while leaving all of the secondary keys present in objects. Then, the objects can be scanned and a follow-up step can remove the keys.

Index creation and deletion represent additional situations where SLIK permits temporary inconsistencies in its implementation, but those internal inconsistencies do not result in inconsistent behavior for applications.

2.8.2 Live Index Split and Migration

Indexlets need to be reconfigurable – we should be able to split one if it gets too large, or migrate one from one server to another. This requires moving index data in bulk from one server to another, which could take a significant amount of time. In the case of migration, the entire indexlet is moved; for splitting, a part of the indexlet is moved.

A straightforward approach would be to lock the indexlet, copy the relevant part to another server, and then unlock. However, this blocks out users from accessing this indexlet and any objects in the data table indexed by it, for the entire duration of this operation.

SLIK uses a different approach: it allows other operations to proceed concurrently on an indexlet that is being copied to another server. SLIK keeps track of the mutations that have occured since the copying started (in a log), and transfers these over as well. A lock is then required only for a short duration of time, while copying over the last mutation. This is similar to approaches used in the past for applications such as virtual machine migration [22] and process migration [30].

3 Implementation

To help us better understand SLIK's design decisions, we implemented it on RAMCloud [24]. RAMCloud is a distributed in-memory key-value storage system and has some important properties that make it a good platform for implementing SLIK. RAMCloud is designed for large-scale applications: this helps us understand if SLIK's architecture can be used for such applications as well. Further, RAMCloud is designed to operate at lowest possible latency by keeping all data in DRAM and using high performance networking: this allows to see whether SLIK's design is efficient enough to operate in ultra-low latency environments. Finally, RAMCloud is open-source and available freely [9]. This has allowed us to make SLIK available freely in open-source format since the inception of the project.

In the previous section, we described the implementation-independent design and architecture of SLIK. In this section, we describe how SLIK was implemented in the context of RAMCloud.

3.1 Overview of RAMCloud

RAMCloud [24] is a storage system that aggregates the memories of thousands of servers into a single coherent key-value store (Figure 3). It offers remote read times of 4.7 µs and write times of 13.5 µs for small objects.

Each storage server contains two components. A *master* module handles read and write requests from the clients. It manages the main memory of the server in a log-structured fashion to store the objects in tables [27].

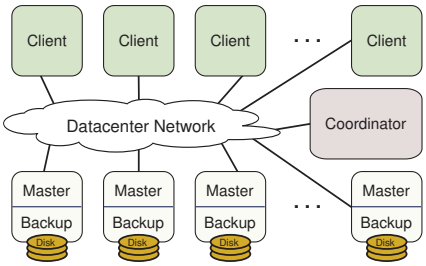


Figure 3: RAMCloud cluster architecture.

A *backup* module uses local disk or flash memory to store backup copies of log information for crash recovery [23]. Each master's log is divided into small segments, and the master scatters multiple replicas of each segment across the available backups. This allows a master's data to be reconstructed within 1-2 s after a crash. RAMCloud uses a small amount of non-volatile memory on each backup, which allows it to declare writes durable as soon as updates have been received by backups, without waiting for disk I/O.

The masters and backups are managed by a central *coordinator* that handles configuration-related issues. The coordinator is a highly reliable and available system (with active and standby instances), but is not normally involved in operations other than those querying or modifying configuration information.

3.2 Implementing Coordination Service for Secondary Indexing

We modified the RAMCloud coordinator to also store and disseminate the metadata about index structures and the servers on which indexlets are stored. When a client starts, it queries the coordinator for the configuration and caches it locally. If this cached configuration becomes stale, the client library discovers this when it sends a query to a server that no longer stores the desired infomation. The client then flushes the local configuration for that table from its cache and fetches up-to-date information from the coordinator (this is described in further detail in [24]).

3.3 Recovering Indexes after Crashes

SLIK stores each indexlet in a RAMCloud table (Section 2.6). Since RAMCloud can recover lost tablets within 1-2 seconds after server crash [23], this ensures that indexes can also be recovered quickly. However, RAMCloud can achieve 1-2 *s* crash recovery only for tablets that are smaller than 500 MB in size. For tablets that are larger than this, RAMCloud will split the tablet during recovery and assign each sub-tablet to a different server, so all of the lost data can be recovered quickly. However, such splitting cannot be used for indexlet backing tables as the B+ tree structure requires all of the objects in the backing table to be present on a single server. Thus, to ensure fast indexlet recovery, SLIK ensures that indexlets are no larger than 500 MB in size. It does this by carrying out live splitting and migration of indexlets that grow beyond the threshold.

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash	2x Crucial M4 SSDs
Disks	CT128M4SSD2 (128 GB)
NIC	Mellanox ConnectX-2 InfiniBand HCA
Switch	Mellanox SX6036 (4X FDR)

Table 3: The server hardware configuration used for benchmarking. All nodes ran Linux 3.16.0 and were connected to a two-level InfiniBand switching fabric.

3.4 Using Log Structured Memory

Our implementation leverages RAMCloud's log-structured approach of storing data to simplify its implementation. First, it uses this log to implement atomicity for multi-node updates discussed in Section 2.7. Second, it uses the log to keep track of the mutations during an index split and/or migration discussed in Section 2.8.2. More concretely, it migrates the relevant data from an indexlet by scanning the log on that server. When it reaches the head of the log, it locks the log head to migrate the last changes (if any).

4 Evaluation

We evaluated the RAMCloud implementation of SLIK to answer the following questions:

- Does SLIK provide low latency? Is it efficient enough to perform index operations at latencies comparable to other RAMCloud operations?
- Is SLIK scalable? Does its performance scale as the number of servers increases?
- How does the scalability of independent partitioning compare to that of colocation?
- How does the performance of indexing with SLIK compare to other state-of-the-art systems?

We chose H-Store [19] for comparison with SLIK because H-Store and VoltDB (which is H-Store's commercial sibling) are in-memory database systems that are becoming widely adopted. We tuned H-Store for each test to get best performance with assistance from H-Store developers [25]. H-Store uses the indexlet/tablet colocation approach to partitioning, so a column can be specified such that all data gets partitioned according to that column. We evaluated H-Store with multiple data partitioning schemes where applicable.

We ran all experiments on an 80-node cluster of identical commodity servers (see Table 3).

4.1 Latency

We first evaluate the latency of basic index operations (lookups and overwrites) using a table with a single

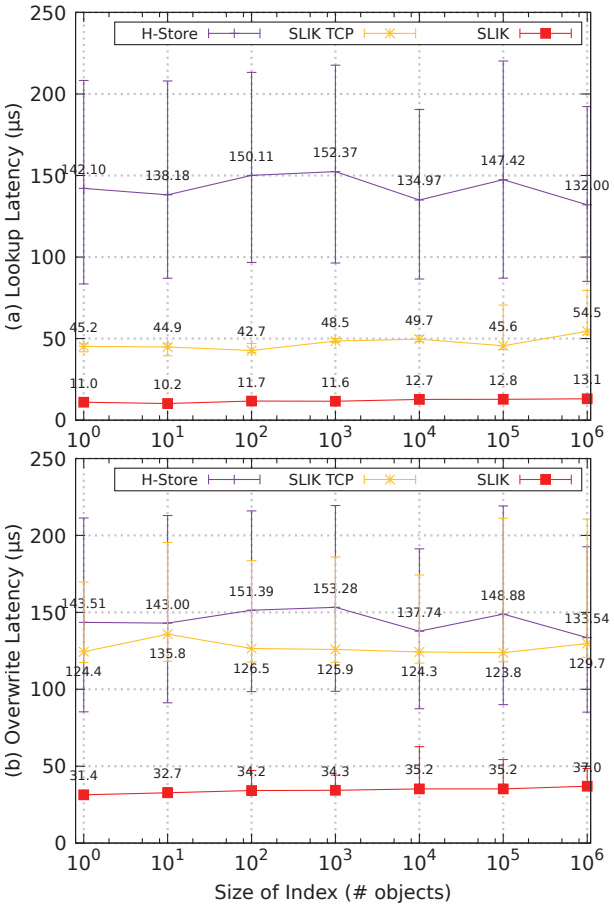


Figure 4: Latency of basic operations as index size increases: (a) read a single object using a secondary key; (b) overwrite an existing object. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements.

secondary index. We then evaluate the latency of object overwrites as the number of secondary indexes increases. We don't evaluate the latency of lookups in this case as it is independent of the number of indexes.

4.1.1 Basic Latency

Figure 4 graphs the latency for single-object index operations on a log scale. The measurements were done with a single client accessing a single table, where each object has a 30 *B* primary key, 30 *B* secondary key and a 100 *B* value. The secondary key has an index (with a single indexlet) corresponding to it.

SLIK lookup: The median time for an index lookup that returns a single object is about 11 µ*s* for a small index and 13.1 µ*s* for an index with a million entries. An index lookup issues two RPCs that read data sequentially (as discussed in Section 2.2) – the time for a non-indexed read in RAMCloud is about 5 µ*s*, making the minimum time required for an index lookup to be 10 µ*s*. The rest of the time is accounted for by the B+ tree lookup time.

SLIK overwrite: The median time for `overwrite`

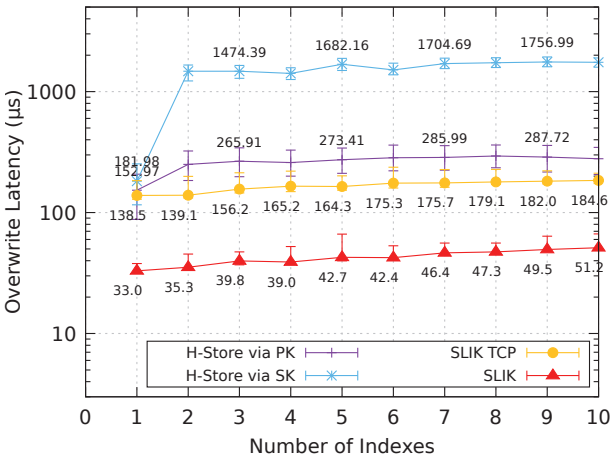


Figure 5: Latency of overwrites as a function of the number of secondary indexes. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements. For H-Store's line *via Pk*, it was partitioned by the primary key and for the line *via Sk*, it was partitioned by the first secondary key. In both the cases, overwrites were done by querying via the primary key. The *y* axis uses a log scale.

ranges from 31.4 µ*s* to 37 µ*s* depending on index size. This is the total cost for doing two sequential durable writes: the first to the index and the second to the object (as discussed in Section 2.3). The removal of old index entries is handled in the background after the overwrite RPC returns to the client.

Comparison: In this benchmark, H-Store is run on a single server so that it uses a single partition for its data and index. It executes all reads and writes locally and no data needs to be transferred to other servers. SLIK provides 3-way distributed replication of objects and index entries to durable backups, whereas H-Store does not perform replication and the durability is disabled. SLIK is about 10x faster than H-Store for lookups and about 4x faster for overwrites.

SLIK is designed to introduce minimal overheads so that it can harness the benefits of low-latency networks and kernel bypass (via InfiniBand). However, we also performed this benchmark by running SLIK with the same network as H-Store: TCP over the InfiniBand network (without kernel bypass). Even in this configuration, SLIK is considerably faster than H-Store for lookups. For overwrites, SLIK provides similar latency as H-Store, but it does so while providing 3-way distributed replication of all data.

4.1.2 Impact of Multiple Secondary Indexes on Overwrite Latency

Figure 5 graphs the latency for overwriting an object as the number of secondary indexes increases. The measurements were done with a single client accessing a single table with 1M objects, where each object has a

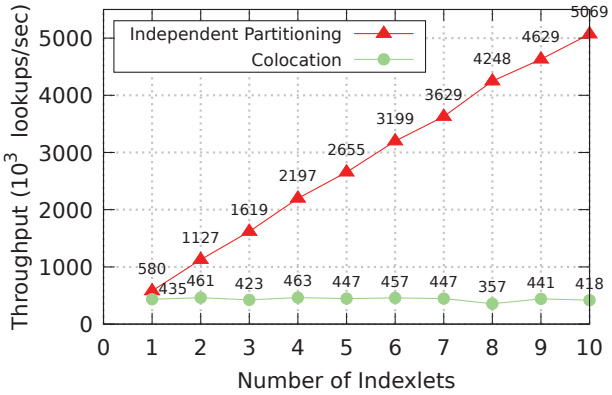


Figure 6: Total index lookup throughput when a single index is divided into multiple indexlets on different servers and queried via multiple clients.

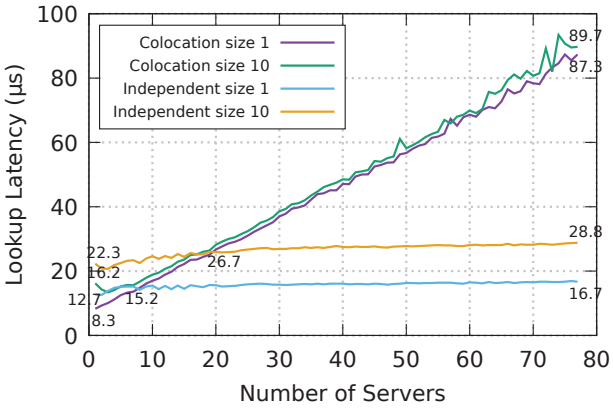


Figure 7: Latency for index lookup when a single index is divided into multiple indexlets on different servers. The size refers to the number of objects returned by a lookup.

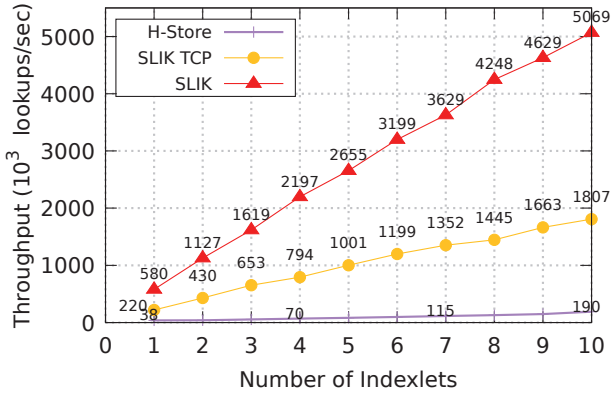


Figure 8: Total index lookup throughput when a single index is divided into multiple indexlets on different servers and queried via multiple clients.

30 *B* primary key, a varying number of 30 *B* secondary keys, and a 100 *B* value. For SLIK, each secondary index has a single partition and is located on a different server.

SLIK: The latency increases moderately for tables with more secondary indexes: overwrites take 32.4 µ*s* with 1 secondary index and 49.8 µ*s* with 10 secondary indexes (about a 50% increase). There is an increase because each of the indexes is stored on a separate server and all the servers must be contacted during writes.

Comparison: SLIK performs better, out of the box without any tuning, while providing durability and replication, than a tuned version of H-Store without durability or replication. For each data point, SLIK and H-Store are both allocated the same number of servers as the number of indexes. H-Store partitions all the data and indexes across these servers. For the line *via PK*, the partitioning is done based on the primary key and for the line *via Sk*, the partitioning is done via the first secondary key. The performance while updating using the same key that is used to partition all data (line *via PK*) is lower than the latency for updates done using a key that was not used for partitioning (line *via Sk*).

4.2 Scalability

One of our goals is to provide scalable performance as the number of servers increases. Given our choice of independent partitioning, we expect a linear increase in throughput as the number of servers increases, since there are no interactions or dependencies between indexlet servers. We also expect minimal impact on latencies as the number of indexlets increase.

To test this hypothesis, we evaluated scalability along two parameters. The first measures the end-to-end throughput of index lookup as the number of indexlets increases. This experiment uses a single table where each object has a 30 *B* primary key, 30 *B* secondary key and 100 *B* value. The index corresponding to the secondary key is divided into a varying number of indexlets, and the table is divided into the same number of tablets: each is stored on a different server. For each data point, the number of clients performing lookups and the number of concurrent lookups per client is varied to achieve the maximum throughput for each system. Each request chooses a random key uniformly distributed across indexlets and returns a matching object. H-Store is partitioned based on the key used for lookups, which is its best configuration for this use case.

The second measures the end-to-end latency of index lookup as the number of indexlets increases. The setup for this experiment is the same as the previous one, except that a single client is used (instead of many), which issues one request at a time in order to expose the latency for each operation.

We first ran these experiments to compare the scalability of the colocation and independent partitioning approaches while keeping everything else the same (Figures 6, 7). We also ran these experiments to evaluate the scalability of SLIK and compare performance with

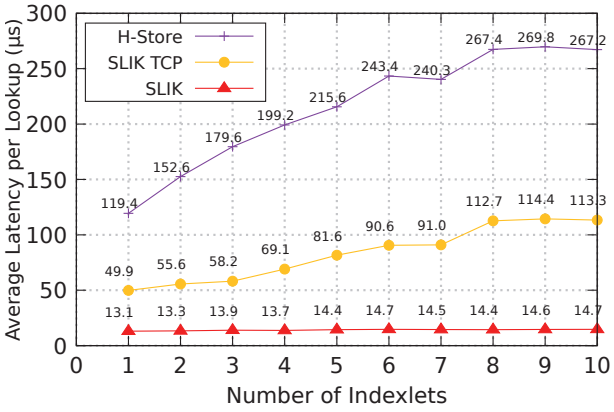


Figure 9: Latency for index lookup when a single index is divided into multiple indexlets on different servers.

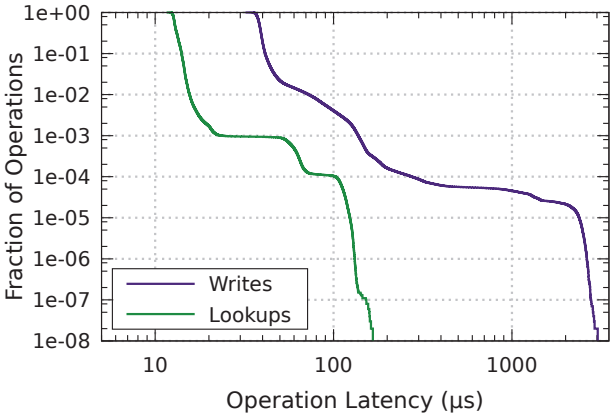


Figure 10: Tail latency distribution for index lookup and `write` operations in SLIK, shown as a reverse CDF with a log scale. A point (x, y) indicates that yth fraction of the 100 M operations measured take at least $x\mu s$ to complete.

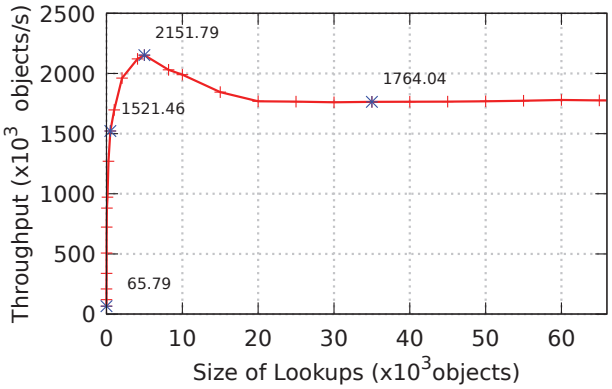


Figure 11: Throughput of index lookup measured by a single client as a function of the total number of objects returned for that lookup.

H-Store (Figures 8, 9).

4.2.1 Independent Partitioning vs Colocation

We first compare the scalability of independent partitioning with the colocation approach using the setups described earlier. For independent partitioning, we used our implementation of SLIK in RAMCloud. For colocation approach, we used the same implementation and changed the partitioning code to use the colocation approach instead.

These figures confirm that independent partitioning performs better at scale. Figure 6 shows that with independent partitioning, the total lookup throughput increases with the addition of servers, whereas with colocation it does not. Figure 7 shows that as the scale gets larger, the cost of doing two sequential RPCs with independent partitioning is lower than a large number of parallel RPCs with colocation.

4.2.2 System Scalability

We then evaluate how SLIK performs at large scale and also compare against H-Store. Figure 8 graphs the end-to-end throughput of index lookup in SLIK and shows that it scales linearly as the number of indexlets is increased. The throughput for H-Store increases sublinearly with the number of partitions. Figure 9 shows that SLIK's index lookup latency has minimal impact as the number of indexlets is increased, while the latency for H-Store increases because each index lookup must contact all indexlet servers.

4.3 Miscellaneous Benchmarks

4.3.1 Tail Latency

Figure 10 graphs the reverse CDFs for single-object lookup and write operations. A single client performed 100 million reads and overwrites on a table with a million objects (where each object has a 30 *B* primary key, 30 *B* secondary key and 100 *B* value) and there is an index corresponding to the secondary key. The index lookup operations have a median latency of about 15 μs, and `write` operations have a median latency of about 36 μs.

4.3.2 Range Lookups

Figure 11 graphs the throughput for index lookup as the total number of objects returned in that lookup increases. The experimental setup is the same as the previous experiment. The total throughput increases as the size of lookup is increased, it peaks at about $2M$ objects/s, and stabilizes at around $1.7M$ objects/s.

5 Related Work

Data storage systems make tradeoffs between various goals: providing higher level data models (like indexing), consistency, durability, scalability and low latency.

Some systems give up certain features in order to optimize for others. For example, MICA [21] is a scalable in-memory key-value store optimized for high throughput; however it does not provide data durability. FaRM [16]

is a main memory distributed computing platform that provides low latency and high throughput by exploiting RDMA; however it does not support secondary indexing.

Some systems support consistent and durable secondary indexes but have higher latency than SLIK. Cassandra [20], DynamoDB [3] and Phoenix [7] on HBase [4] provide local secondary indexes which are partitioned using the colocation approach. While these indexes provide high consistency, they also have higher latency as each request needs to contact all the servers (as described in Section 2.2). STI-BT [15] extends an existing key-value store to provide scalable and consistent indexing, and F1 [28] extends Spanner [14] to provide a distributed relational database; however they also have similarly high latencies.

Some of the systems above like DynamoDB [3] and Phoenix [7] on HBase [4] also provide global secondary indexes, but they are only eventually consistent. Moreover, a query on an index can return only those attributes of the object data that have been projected onto that index by the developer and stored with it.

Many other systems provide weak consistency guarantees, while still having latencies comparable to systems above: CouchDB [2] is eventually consistent; PNUTS [13], Espresso [26] and Tao [12] have weak consistency guarantees.

RAMP [10] proposes a new consistency model for transactions called Read Atomic Isolation which can be used to enable strong consistency between object and index updates in a distributed storage system. It proposes three algorithms that offer different trade-offs between speed and the amount of metadata required. The fastest version of RAMP requires two serialized round-trips for writes, which is the same as SLIK but requires a comparatively large amount of metadata that needs to be stored and transported over the network.

H-Store [19] is a main-memory distributed storage system that also provides consistent indexing at a large scale. It partitions data based on a specified attribute (which can be a primary key or a secondary key), which helps the queries based on the partitioning column benefit from the data locality. However, all queries using other attributes need to contact all the partitions to fetch the result, which adversely impacts its performance.

HyperDex [17] is a disk-based large-scale storage system that supports consistent indexing. It partitions data using a novel hyperspace hashing scheme by mapping objects' attributes into a multidimentional space. As the number of attributes increase, the number of hyperspaces increases dramatically. HyperDex alleviates this by partitioning tables with many attributes into multiple lower-dimensional hyperspaces called subspaces. HyperDex also replicates the entire contents of objects in each index. This means that while HyperDex provides an efficient mechanism for search, it uses more storage space for the extra copies of objects. While this is acceptable for disk based systems, it would be very expensive for main-memory based systems.

We have compared approaches taken by other systems and discussed their tradeoffs with the approaches adopted by SLIK in Section 2. Further, in Section 4 we compared SLIK performance with H-Store [19] and found that SLIK outperformed it by a large factor. We also benchmarked HyperDex [17]. However, we omit these benchmarks due to space constraints and because it was hard to quantify how much of its poorer performace was due to its use of disk for storage.

SLIK's most unique aspect is its combination of low latency and consistency at large scale; other systems sacrifice at least one of these.

6 Conclusion

We have shown that it is possible to have durable and consistent secondary indexes in a key-value storage system at extremely low latency and large scale. We made design decisions by considering tradeoffs between various approaches or by developing new algorithms where acceptable solutions did not exist. This design provides secondary indexing that provides better scalability and latency than existing systems, without any tuning for specific use cases.

Modern scalable storage systems need not sacrifice the powerful programming model provided by traditional relational databases. Furthermore, when implemented using DRAM-based storage and state-of-the-art networking, storage systems can provide unprecedented performance. SLIK is an important step on the path to a high-function, low-latency, large-scale storage system.

7 Acknowledgments

This work was supported by C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA), and by grants from Emulex, Facebook, Google, Huawei, Inventec, NEC, NetApp, Samsung, and VMware.

We would like to thank Hector Garcia-Molina and Keith Winstein for their guidance. We would also like to thank Jonathan Ellithorpe, Greg Hill, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Mendel Rosenblum, Steve Rumble, William Sheu, Ryan Stutsman, and Henry Qin for interesting discussions and feedback at various points in the project. Finally, we would like to thank the anonymous reviewers and our shepherd, Sriram Rao, for their helpful comments.

References

[1] Aerospike. http://www.aerospike.com/.

[2] CouchDB. http://couchdb.apache.org/.

[3] DynamoDB. http://aws.amazon.com/documentation/dynamodb/.

[4] HBase. http://hbase.apache.org/.

[5] MongoDB. http://www.mongodb.org/.

[6] MySQL InnoDB Storage Engine. http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html.

[7] Phoenix. http://phoenix.apache.org/.

[8] Redis. http://www.redis.io/.

[9] RAMCloud Git Repository, 2015. https://github.com/PlatformLab/RAMCloud.git.

[10] BAILIS, P., FEKETE, A., HELLERSTEIN, J. M., GHODSI, A., AND STOICA, I. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 27–38.

[11] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011), vol. 11, pp. 223–234.

[12] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., ET AL. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference* (2013), pp. 49–60.

[13] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[14] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ET AL. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[15] DIEGUES, N., AND ROMANO, P. Sti-bt: A scalable transactional index. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on* (2014), IEEE, pp. 104–113.

[16] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI* (2014), vol. 14.

[17] ESCRIVA, R., WONG, B., AND SIRER, E. G. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 25–36.

[18] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *OSDI* (2010), pp. 61–74.

[19] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1496–1499.

[20] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[21] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014), 36.

[22] NELSON, M., LIM, B.-H., HUTCHINS, G., ET AL. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track* (2005), pp. 391–394.

[23] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 29–41.

[24] OUSTERHOUT, J., GOPALAN, A., GUPTA, A., KEJRIWAL, A., LEE, C., MONTAZERI, B., ONGARO, D., PARK, S. J., QIN, H., ROSENBLUM, M., RUMBLE, S., STUTSMAN, R., AND YANG, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[25] PAVLO, A. Personal communications, March 17 2015.

[26] QIAO, L., SURLAKER, K., DAS, S., QUIGGLE, T., SCHULMAN, B., GHOSH, B., CURTIS, A., SEELIGER, O., ZHANG, Z., AURADAR, A., BEAVER, C., BRANDT, G., GANDHI, M., GOPALAKRISHNA, K., IP, W., JGADISH, S., LU, S., PACHEV, A., RAMESH, A., SEBASTIAN, A., SHANBHAG, R., SUBRAMANIAM, S., SUN, Y., TOPIWALA, S., TRAN, C., WESTERMAN, J., AND ZHANG, D. On brewing fresh Espresso: LinkedIn's distributed data serving platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 1135–1146.

[27] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), pp. 1–16.

[28] SHUTE, J., OANCEA, M., ELLNER, S., HANDY, B., ROLLINS, E., SAMWEL, B., VINGRALEK, R., WHIPKEY, C., CHEN, X., JEGERLEHNER, B., ET AL. F1: The fault-tolerant distributed RDBMS supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 777–778.

[29] STUTSMAN, R., LEE, C., AND OUSTERHOUT, J. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 17–30.

[30] ZAYAS, E. Attacking the process migration bottleneck. *ACM SIGOPS Operating Systems Review 21*, 5 (1987), 13–24.

Understanding Manycore Scalability of File Systems

Changwoo Min Sanidhya Kashyap Steffen Maass Woonhak Kang Taesoo Kim
Georgia Institute of Technology

Abstract

We analyze the manycore scalability of five widely-deployed file systems, namely, `ext4`, `XFS`, `btrfs`, `F2FS`, and `tmpfs`, by using our open source benchmark suite, FxMark. FxMark implements 19 microbenchmarks to stress specific components of each file system and includes three application benchmarks to measure the macroscopic scalability behavior. We observe that file systems are hidden scalability bottlenecks in many I/O-intensive applications even when there is no apparent contention at the application level. We found 25 scalability bottlenecks in file systems, many of which are unexpected or counterintuitive. We draw a set of observations on file system scalability behavior and unveil several core aspects of file system design that systems researchers must address.

1 Introduction

Parallelizing I/O operations is a key technique to improve the I/O performance of applications [46]. Today, nearly all applications implement concurrent I/O operations, ranging from mobile [52] to desktop [46], relational databases [10], and NoSQL databases [6, 43]. There are even system-wide movements to support concurrent I/O efficiently for applications. For example, commercial UNIX systems such as HP-UX, AIX, and Solaris extended the POSIX interface [48, 50, 68], and open source OSes like Linux added new system calls to perform asynchronous I/O operations [17].

Two recent technology trends indicate that parallelizing I/O operations will be more prevalent and pivotal in achieving high, scalable performance for applications. First, storage devices have become significantly faster. For instance, a single NVMe device can handle 1 million IOPS [14, 72, 86], which roughly translates to using 3.5 processor cores to fully drive a single NVMe device [51]. Second, the number of CPU cores continues to increase [19, 62] and large, high-performance databases have started to embrace manycore in their core operations [9, 11, 44, 49, 53, 90]. These trends seem to promise an implicit scaling of applications already employing concurrent I/O operations.

In this paper, we first question the practice of *concurrent I/O* and understand the *manycore scalability*[1] behavior, that we often take for granted. In fact, this is the right moment to thoroughly evaluate the manycore scalability of file systems, as many applications have started hitting this wall. Moreover, most of the critical design decisions are "typical of an 80's era file system" [37]. Recent studies on manycore scalability of OS typically use memory file systems, e.g., `tmpfs`, to circumvent the effect of I/O operations [16, 20–22, 32–34]. So, there has been no in-depth analysis on the manycore scalability of file systems. In most cases, when I/O performance becomes a scalability bottleneck without saturating disk bandwidth, it is not clear if it is due to file systems, its usage of file systems, or other I/O subsystems.

Of course, it is difficult to have the complete picture of file system scalability. Nevertheless, in an attempt to shed some light on it, we present an extensive study of manycore scalability on file systems. To evaluate the scalability aspects, we design and implement the FxMark benchmark suite, comprising 19 microbenchmarks stressing each building block of a file system, and three application benchmarks representing popular I/O-intensive tasks (i.e., mail server, NoSQL key/value store, and file server). We analyze five popular file systems in Linux, namely, `ext4`, `XFS`, `btrfs`, `F2FS`, and `tmpfs`, on three storage mediums: `RAMDISK`, `SSD`, and `HDD`.

Our analysis revealed unanticipated scalability behavior that should be considered while designing I/O-intensive applications. For example, all operations on a directory are sequential regardless of read or write; a file cannot be concurrently updated even if there is no overlap in each update. Moreover, we should revisit the core design of file systems for manycore scalability. For example, the consistency mechanisms like journaling (`ext4`), copy-on-write (`btrfs`), and log-structured writing (`F2FS`) are not scalable. We believe that FxMark is effective in two ways. It can identify the manycore scalability problems in existing file systems and further guide and evaluate the new design of a scalable file system in the future.

In summary, we make the following contributions:

- We design an open source benchmark suite, FxMark, to evaluate the manycore scalability of file systems. The FxMark benchmark suite is publicly available at `https://github.com/sslab-gatech/fxmark`.
- We evaluate five widely-used Linux file systems on an 80-core machine with FxMark. We also analyze how the design of each file system and the `VFS` layer affect their scalability behavior.
- We summarize our insights from the identified scalability bottlenecks to design a new scalable file system for the future.

The rest of this paper is organized as follows: §2 pro-

[1] We sometimes mention manycore scalability as scalability for short.

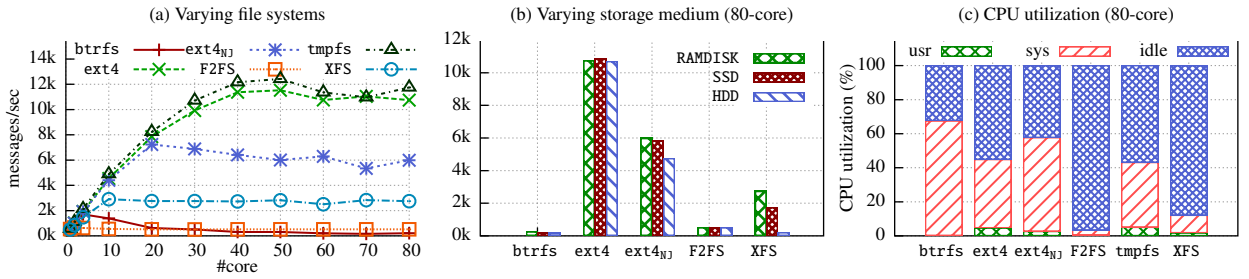


Figure 1: Exim throughput (i.e., delivering messages) on six file systems (i.e., `btrfs`, `ext4`, `F2FS`, `tmpfs`, `XFS`, and `ext4` without journaling, $\texttt{ext4}_{\texttt{NJ}}$) and three storage mediums (i.e., `RAMDISK`, `SSD`, `HDD` at 80-core). We found that the manycore scalability of Exim depends a lot on the file systems (e.g., `ext4` is 54× faster than `btrfs` at 80-core), but does not depend much on storage mediums (e.g., marginal performance difference of `ext4` on `RAMDISK` and `HDD`). To avoid known scalability bottlenecks in Exim, we modified Exim as in the previous study [21] and configured it to disable per-message `fsync()` call.

vides the motivation for our work with a case study; §3 provides an overview of FxMark, and §4 describes our evaluation strategies; §5 and §6 show analysis results; §7 summarizes our findings and §8 discusses implications for OS and file system designers; §9 compares our work with previous work; Finally, §10 provides the conclusion.

2 Case Study

In this section, we show how non-scalable file systems can break the scalability of embarrassingly parallel (i.e., ideally parallelizable) I/O-intensive applications in unexpected ways. The Exim [8] email server is one such application that is designed to scale perfectly, at least from an application's perspective. For example, Exim delivers messages to appropriate mail boxes in parallel and performs each delivery independently. In fact, the message delivery heavily utilizes I/O operations. It consists of a series of operations ranging from creating, renaming, and deleting small files in the spool directories to appending the message body to the per-user mail file. Unfortunately, Exim fails to scale over 10, 20, or 40-core, at most, among the popular file systems in Linux as shown in Figure 1.

File systems kill application scalability. Figure 1(a) shows the throughput of Exim with six different file systems on `RAMDISK`. Surprisingly, the scalability and performance of Exim are significantly dependent on the file system. The performance gap between the best file system, `tmpfs`, and the worst, `btrfs`, is 54× at 80-core. `ext4` and `tmpfs` scale linearly up to 40-core; then the Linux kernel becomes the bottleneck. However, Exim on `F2FS` is hardly scalable; it is 21× slower than `ext4` at 80-core.

Faster storage mediums do not guarantee scalability. With a reasonably large memory, the page cache will absorb most read and write operations, and most write operations can be performed in the background. In this case, the in-memory structures in the file systems determine the scalability, rather than the storage medium. Figure 1(b) shows that all file systems, except `XFS`, show a marginal performance difference among `RAMDISK`, `SSD`, and `HDD` at 80-core. In this case, performance with `XFS` is largely affected by the storage medium since `XFS` mostly waits for flushing journal data to disk due to its heavy metadata update operations.

Fine-grained locks often impede scalability. We may assume that the worst performing file system, `btrfs`, will be mostly in idle state, since it is waiting for events from the storage. On the contrary, Figure 1(c) shows that 67% of CPU time is spent in the kernel mode for `btrfs`. In particular, `btrfs` spends 47% of CPU time on synchronization to update its root node. In a common path without any contention, `btrfs` executes at least 10 atomic instructions to acquire a single B-tree node lock (`btrfs_tree_lock()`) and it must acquire locks of all interim nodes from a leaf to the root. If multiple readers or writers are contending to lock a node, each thread *retries* this process. Under heavy contention, it is typical to retry *a few hundreds times* to lock a single node. These frequent, concurrent accesses to synchronization objects result in a performance collapse after 4-core, as there is no upper bound on atomic instructions for updating the root node.

Subtle contentions matter. Figure 1(a) shows another anomaly with $\texttt{ext4}_{\texttt{NJ}}$ (i.e., `ext4` with no journaling) performing 44% *slower* than `ext4` itself. We found that two independent locks (i.e., a spinlock for journaling and a mutex for directory update) interleave in an unintended fashion. Upon `create()`, `ext4` first hits the journal spinlock (`start_this_handle()`) for metadata consistency and then hits the parent directory mutex (`path_openat()`) to add a new inode to its parent directory. In `ext4`, the serialization coordinated by the journal spinlock incurs little contention while attempting to hold the directory mutex. On the contrary, the contending directory mutex in $\texttt{ext4}_{\texttt{NJ}}$ results in expensive side-effects, such as sleeping on a waiting queue after a short period of opportunistic spinning.

Speculating scalability is precarious. The Exim case study shows that it is difficult for application developers or even file systems developers to speculate on the scalability of file system implementations. To identify such scalability problems in file systems, our community needs a proper benchmark suite to constantly evaluate and guide the design of file systems for scalability.

Type	Mode	Operation	Sharing Level		
			LOW	MEDIUM	HIGH
DATA	READ	BLOCK READ	✓	✓	✓
	WRITE	OVERWRITE	✓	✓	-
		APPEND	✓	-	-
		TRUNCATE	✓	-	-
		SYNC	✓	-	-
META	READ	PATH NAME READ	✓	✓	✓
		DIRECTORY LIST	✓	✓	-
	WRITE	CREATE	✓	✓	-
		UNLINK	✓	✓	-
		RENAME	✓	✓	-

Table 1: Coverage of the FXMARK microbenchmarks. FXMARK consists of 19 microbenchmarks that we categorize based on four criteria: data types, modes, operations, and sharing levels that are accordingly represented in each column on the table.

3 FXMARK Benchmark Suite

There are 19 microbenchmarks in FXMARK that are designed for systematically identifying scalability bottlenecks, and three well-known I/O-intensive application benchmarks to reason about the scalability bottlenecks in I/O-intensive applications.

3.1 Microbenchmarks

Exploring and identifying scalability bottlenecks *systematically* is difficult. The previous studies [21, 22, 32–34] on manycore scalability show that most applications are usually stuck with a few bottlenecks, and resolving them reveals the next level of bottlenecks.

To identify scalability problems in file system implementations, we designed microbenchmarks stressing seven different components of file systems: (1) path name resolution, (2) page cache for buffered I/O, (3) inode management, (4) disk block management, (5) file offset to disk block mapping, (6) directory management, and (7) consistency guarantee mechanism.

Table 1 illustrates the way FXMARK categorizes each of these 19 microbenchmarks with varying stressed data types (i.e., file data or file system metadata), a related file system operation (e.g., `open()`, `create()`, `unlink()`, etc.), and its contention level (i.e., low, medium and high).

A higher contending level means the microbenchmark shares a larger amount of common code with the increasing number of cores, marked as *sharing level* for clarity.

For reading data blocks, FXMARK provides three benchmarks based on the sharing level: (1) reading a data block in the *private file* of a benchmark process (low), (2) reading a *private data block* in the *shared file* among benchmark processes (medium), and (3) reading the *same data block* in the shared file (high). As a convention, we denote each microbenchmark with four letters representing type, mode, etc. For instance, we denote three previous examples with `DRBL` (i.e., Data, Read, Block read, and Low), `DRBM`, and `DRBH`, respectively. Table 2 shows a detailed description of each benchmark, including its core operation and expected contention.

To measure the scalability behavior, each benchmark runs its file system-related operations as a separate process pinned to a core; for example, each benchmark runs up to 80 physical cores to measure the scalability characteristics. Note that we use processes instead of threads, to avoid a few known scalability bottlenecks (e.g., allocating file descriptors and virtual memory management) in the Linux kernel [21, 33]. FXMARK modifies the CPU count using CPU hotplug mechanism [76] in Linux. To minimize the effect of NUMA, CPU cores are assigned on a per socket basis; for example, in the case of 10 cores per socket, the first 10 CPU cores are assigned from the first CPU socket and the second 10 CPU cores are assigned from the second CPU socket. To remove interference between runs, FXMARK formats a testing file system and drops all memory caches (i.e., page, inode, and dentry caches) before each run.

3.2 Application Benchmarks

Although a microbenchmark can precisely pinpoint scalability bottlenecks in file system components, scalability problems in applications might be relevant to only some of the bottlenecks. In this regard, we chose three application scenarios representing widely-used I/O-intensive tasks: mail server, NoSQL database, and file server.

Mail server. Exim is expected to linearly scale with the number of cores, at least from the application's perspective. But as Figure 1 shows, Exim does not scale well even after removing known scalability bottlenecks [21]. To further mitigate scalability bottlenecks caused by the Linux kernel's process management, we removed one of the two process invocations during the message transfer in Exim.

NoSQL database. RocksDB is a NoSQL database and storage engine based on log-structured merge-trees (LSM-trees) [43, 73]. It maintains each level of the LSM-tree as a set of files and performs multi-threaded compaction for performance, which will eventually determine the write performance. We use `db_bench` to benchmark RocksDB using the `overwrite` workload with disabled compression, which overwrites randomly generated key/value-pairs.

File server. To emulate file-server I/O-activity, we use the DBENCH tool [5]. The workload performs a sequence of create, delete, append, read, write, and attribute-change operations, with a specified number of clients processing the workload in parallel.

4 Diagnosis Methodology

4.1 Target File Systems

We chose four widely-used, disk-based file systems and one memory-based file system: `ext4`, `XFS`, `btrfs`, `F2FS`, and `tmpfs`.

T	M	Name	Operation	Description	Expected Contention
DATA	READ	DRBL	pread("$ID/data",b,4K,0)	Read a block in a private file	None
		DRBM	pread("share",b,4K,$ID*4K)	Read a private block in a shared file	Shared file accesses
		DRBH	pread("share",b,4K,0)	Read a shared block in a shared file	Shared block accesses
	WRITE	DWOL	pwrite("$ID/data",b,4K,0)	Overwrite a block in a private file	None
		DWOM	pwrite("share",b,4K,$ID*4K)	Overwrite a private block in a shared file	Updating inode attributes (e.g., m_time)
		DWAL	append("$ID/data",b,4K)	Append a block in a private file	Disk block allocations
		DWTL	truncate("$ID/data",4K)	Truncate a private file to a block	Disk block frees
		DWSL	pwrite("$ID/data",b,4K,0) fsync("$ID/data")	Synchronously overwrite a private file	Consistency mechanism (e.g., journaling)
META	READ	MRPL	close(open("$ID/0/0/0/0"))	Open a private file in five-depth directory	Path name look-ups
		MRPM	close(open("$R/$R/$R/$R/$R"))	Open an arbitrary file in five-depth directory	Path name look-ups
		MRPH	close(open("0/0/0/0/0"))	Open the same file in five-depth directory	Path name look-ups
		MRDL	readdir("$ID/")	Enumerate a private directory	None
		MRDM	readdir("share/")	Enumerate a shared directory	Shared directory accesses
	WRITE	MWCL	create("$ID/$N")	Create an empty file in a private directory	Inode allocations
		MWCM	create("share/$ID.$N")	Create an empty file in a shared directory	Inode allocations and insertions
		MWUL	unlink("$ID/$N")	Unlink an empty file in a private directory	Inode frees
		MWUM	unlink("share/$ID.$N")	Unlink an empty file in a shared directory	Inode frees and deletions
		MWRL	rename("$ID/$N","$ID/$N.2")	Rename a private file in a private directory	None
		MWRM	rename("$ID/$N","share/$ID.$N")	Move a private file to a shared directory	Insertions to the shared directory

NOTE. $ID: a unique ID of a test process b: a pointer to a memory buffer $R: a random integer between 0 and 7 $N: a test iteration count

Table 2: Microbenchmarks in FxMark. Each benchmark is identified by four letters based on type (marked as T), mode (marked as M), operation, and sharing level, as described in Table 1. Each microbenchmark is designed to stress specific building blocks of modern file systems (e.g., journaling and dcache) to evaluate their scalability on manycore systems.

Ext4 is arguably the most popular, mature Linux file system. It inherits well-proven features (e.g., bitmap-based management of inodes and blocks) from Fast File System (FFS) [69]. It also implements modern features such as extent-based mapping, block group, delayed allocation of disk blocks, a hash tree representing a directory, and journaling for consistency guarantee [27, 42, 67]. For journaling, ext4 implements write-ahead logging (as part of JBD2). We use ext4 with journaling in ordered mode and without it, marked as ext4 and ext4_{NJ}, to see its effect on file system scalability.

XFS is designed to support very large file systems with higher capacity and better performance [85]. XFS incorporates B+ trees in its core: inode management, free disk block management, block mapping, directory, etc [83]. However, using B+ trees incurs huge bulk writes due to tree balancing; XFS mitigates this by implementing *delayed logging* to minimize the amount of journal writes: (1) logically log the operations rather than tracking physical changes to the pages and (2) re-log the same log buffer multiple times before committing [30, 31].

Btrfs is a copy-on-write (CoW) file system that represents everything, including file data and file system metadata, in CoW optimized B-trees [77]. Since the root node of such B-trees can uniquely represent the state of the entire file system [78], btrfs can easily support a strong consistency model, called version consistency [29].

F2FS is a flash-optimized file system [23, 60, 63]. It follows the design of a log-structured file system (LFS) [80] that generates a large sequential write stream [55, 70]. Unlike LFS, it avoids the wandering tree problem [15] by updating some of its metadata structures in-place.

Tmpfs is a memory-based file system that works without a backing storage [79]. It is implemented as a simple wrapper for most functions in the VFS layer. Therefore, its scalability behavior should be an ideal upper bound of the Linux file systems' performance that implements extra features on top of VFS.

4.2 Experimental Setup

We performed all of the experiments on our 80-core machine (8-socket, 10-core Intel Xeon E7-8870) equipped with 512 GB DDR3 DRAM. For storage, the machine has both a 1 TB SSD (540 MB/s for reads and 520 MB/s for writes) and a 7200 RPM HDD with a maximum transfer speed of 160 MB/s. We test with Linux kernel version 4.2-rc8. We mount file systems with the noatime option to avoid metadata update for read operations. RAMDISK is created using tmpfs.

4.3 Performance Profiling

While running each benchmark, FxMark collects the CPU utilization for sys, user, idle, and iowait to see a microscopic view of a file system reaction to stressing its components. For example, a high idle time implies that the operation in a microbenchmark impedes the scalability behavior (e.g., waiting on locks); a high iowait time implies that a storage device is a scalability bottleneck. For further analysis, we use perf [7] to profile the entire system and to dynamically probe a few interesting functions (e.g., mutex_lock(), schedule(), etc.) that likely induce such idle time during file system operations.

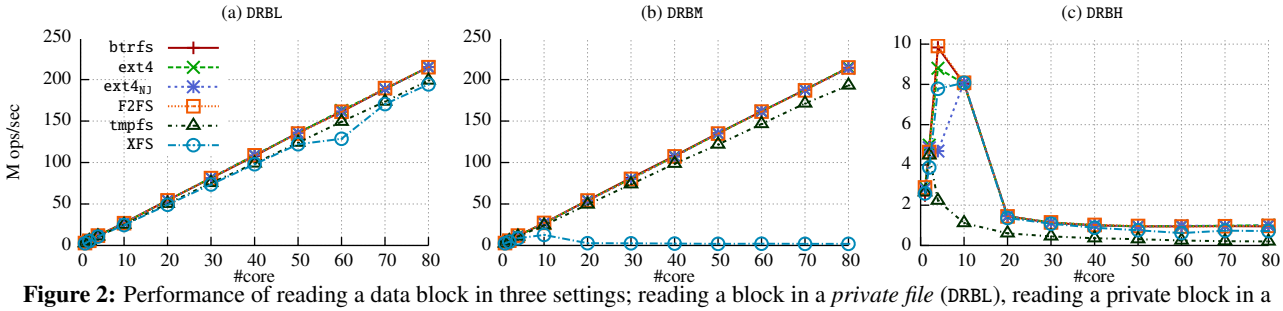


Figure 2: Performance of reading a data block in three settings; reading a block in a *private file* (DRBL), reading a private block in a *shared file* (DRBM), and reading a *shared block* in a shared file (DRBH). All file systems scale linearly in DRBL. XFS fails to scale in DRBM because of the per-inode read/write semaphore. In DRBH, all file systems show their peak performance around 10-core because of contending per-page reference counters in VFS (§5.1.1).

5 Microbenchmark Analysis

In this section, we first describe the analysis results of each microbenchmark in buffered I/O mode starting from simple operations on file data (§5.1) and going to more complicated file system metadata operations (§5.2). Then, we analyze the results of file data operation in direct I/O mode (§5.3). Lastly, we analyze how the performance of the storage medium affects scalability (§5.4).

5.1 Operation on File Data

5.1.1 Block Read

We start from the simplest case: each test process reads a block in its respective private file (DRBL). As Figure 2 shows, all test file systems scale nearly linearly.

However, when each test process reads a private block in the *shared file* (DRBM), we observe that XFS's performance collapses near 10-core. Unlike other file systems, it spends 40% of execution and idle time at per-inode read/write semaphores (fs/xfs/xfs_file.c:329) when running on 80 cores. XFS relies heavily on per-inode read/write semaphores to orchestrate readers and writers in a fine-grained manner [85]. However, the performance degradation does not come from unnecessary waiting in the semaphores. At every read(), XFS acquires and releases a read/write semaphore of a file being accessed in shared mode (down_read() and up_read()). A read/write semaphore internally maintains a reader counter, such that every operation in shared mode updates the reader counter *atomically*. Therefore, concurrent read operations on a shared file in XFS actually perform atomic operations on a shared reader counter. This explains the performance degradation after 10 cores. In fact, at XFS's peak performance, the cycles per instruction (CPI) is 20 at 10-core, but increases to 100 at 20-core due to increased cache line contention on updating a shared reader counter.

For reading the same block (DRBH), all file systems show a performance collapse after 10-core. Also, the performance at 80-core is 13.34× (for tmpfs) lower than that on a single core. We found that more than 50% of the time is being spent on reference counter operations for the page cache. The per-page reference counting is used for concurrency control of a page in a per-inode page cache. The per-page reference counter is also updated using atomic operations so that all test processes contend to update the same cache line. In the case of ext4, the CPI is 14.3 at 4-core but increases to 100 at 20-core due to increased cache-coherence delays.

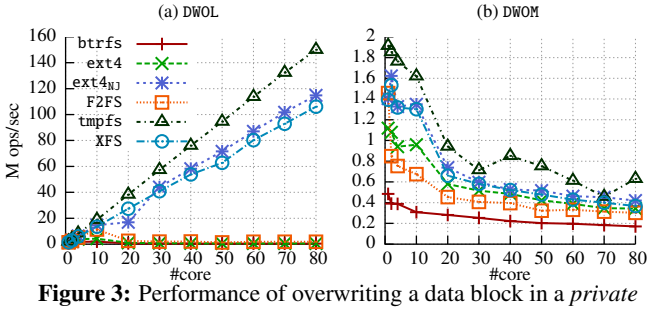


Figure 3: Performance of overwriting a data block in a *private file* (DWOL) and overwriting a *private block* in a shared file (DWOM). In DWOL, although they do not seem to have explicit contention, btrfs, ext4 and F2FS fail to scale due to their consistency mechanisms. Note that XFS is an exception among journaling file systems (see §5.1.2). In DWOM, all file systems fail to scale regardless of consistency mechanisms.

5.1.2 Block Overwrite

At first glance, overwriting a block in a private file (DWOL) should be an ideal, scalable operation; only the private file block and inode features such as last modified time need to be updated. However, as shown in Figure 3, ext4, F2FS, and btrfs fail to scale. In ext4, starting and stopping journaling transactions (e.g., jbd2_journal_start()) in JBD2 impedes its scalability. In particular, acquiring a read lock (journal->j_state_lock) and atomically increasing a counter value (t_handle_count) take 17.2% and 9.4% of CPU time, respectively. Unlike ext4, XFS scales well due to delayed logging [30, 31], which uses logical logging and re-logging to minimize the amount of journal write. In F2FS, write operations eventually trigger segment cleaning (or garbage collection) to reclaim invalid blocks. Segment cleaning freezes all file system operations for checkpointing of the file system status by holding the exclusive lock of a file system-wide read/write semaphore (sbi->cp_rwsem). btrfs is a CoW-based file system that never overwrites a physical block. It allo-

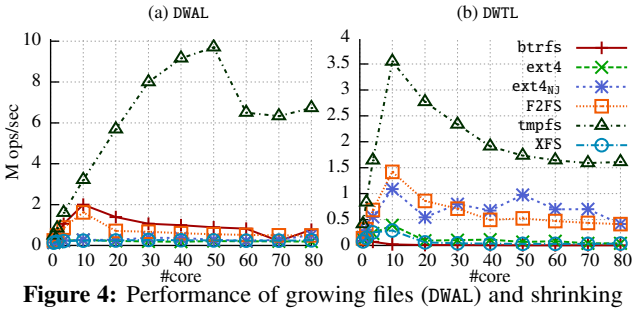


Figure 4: Performance of growing files (`DWAL`) and shrinking files (`DWTL`). None of the tested file systems scales.

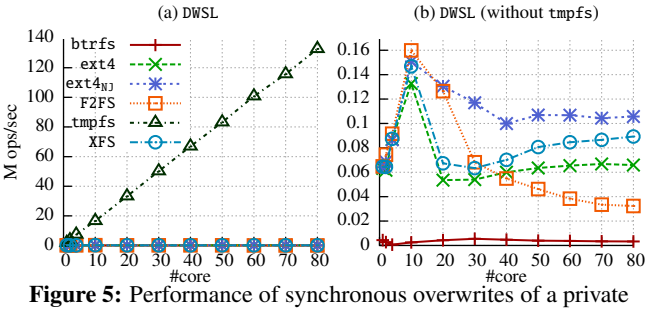


Figure 5: Performance of synchronous overwrites of a private file (`DWSL`). Only `tmpfs` ignoring `fsync()` scales.

cates a new block for every write operation so that its disk block allocation (i.e., updating its extent tree) becomes the sequential bottleneck.

When every test process overwrites each private block in the shared file (`DWOM`), none of the file systems scale. The common bottleneck is an inode mutex (`inode->i_mutex`). The root cause of this sequential bottleneck stems from file system-specific implementations, not `VFS`. None of the tested file systems are implementing a range-based locking mechanism, which is common in parallel file systems [81]. In fact, this is a serious limitation in scaling I/O-intensive applications like DBMS. The common practice of running multiple I/O threads is not effective when applications are accessing a shared file, regardless of the underlying file systems. Furthermore, it may incur a priority inversion between I/O and latency-sensitive client threads [24, 25], potentially disrupting the application's scalability behavior.

5.1.3 File Growing and Shrinking

For file growing and shirking, all disk-based file systems fail to scale after 10-core (`DWAL` in Figure 4). In `F2FS`, allocating or freeing disk blocks entails updating two data structures: SIT (segment information table) tracking block validity, and NAT (node address table) tracking inode and block mapping tables. Contention in updating SIT and NAT limits `F2FS`'s scalability. When allocating blocks, checking disk free-space and updating the NAT consumes 78% of the execution time because of lock contentions (`nm1->nat_tree_lock`). Similarly, there is another lock contention for freeing blocks to invalidate free blocks. This exhausts 82% of the execution time (`sit_i->sentry_lock`). In `btrfs`, when growing a file, the sequential bottleneck is checking and reserving free space (`data_info->lock` and `delalloc_block_rsv->lock`). When shrinking a file, `btrfs` suffers from updating an extent tree, which keeps track of the reference count of disk blocks: A change in reference counts requires updates all the way up to the root node.

In `ext4` and `XFS`, the *delayed allocation* technique, which defers block allocation until writeback to reduce fragmentation of a file, is also effective in improving manycore scalability by reducing the number of block allocation operations. Because of this, the scalability bottleneck of `ext4` and `XFS` is not the block allocation but rather their journaling mechanisms. About 76–96% of the execution time was spent on journaling. `ext4` spends most of its execution time on manipulating `JBD2` shared data structures (e.g., `journal_j_flags`) protected by spinlocks and atomic instructions. `XFS` spends most of its execution time waiting on flushing its log buffers. Upon truncating files, these file systems face the same performance bottleneck.

In `tmpfs`, checking the capacity limit (`__vm_enough_memory()`) becomes a scalability problem. As the used space approaches the capacity limit (at 50-core in this case), the checking takes a slow path for precise comparison of the remaining disk space. Tracking the used space by using a per-CPU counter scales up to 50-core, but fails to scale for more cores because of a contending spinlock in a per-CPU counter on the slow path to get a true value. When freeing blocks, updating per-`cgroup` page usage information using atomic decrements causes a performance collapse after 10-core.

5.1.4 File Sync Operation

When using `fsync()`, a file system synchronously flushes dirty pages of a file and disk caches. In this regard, all file systems can scale up to the limitation of the storage medium. Although we use memory as a storage backend, none of the file systems scale, except `tmpfs`, which ignores `fsync()` operations. Notice that `fsync()` on `btrfs` is significantly slower than other file systems (see Figure 5). Similar to §5.1.2, `btrfs` propagates a block update to its root node so a large number of metadata pages need to be written[2]. All file systems (except for `tmpfs` and `btrfs`) start to degrade after 10-core. That is due to locks protecting flush operations (e.g., `journal->j_state_lock`), which start contending after roughly 10-core.

[2] To minimize `fsync()` latency, `btrfs` maintains a special *log-tree* to defer checkpointing the entire file system until the log is full. In the case of `fsync()`-heavy workloads, like `DWSL`, the log quickly becomes full; therefore, checkpointing or updating the root node becomes a sequential bottleneck.

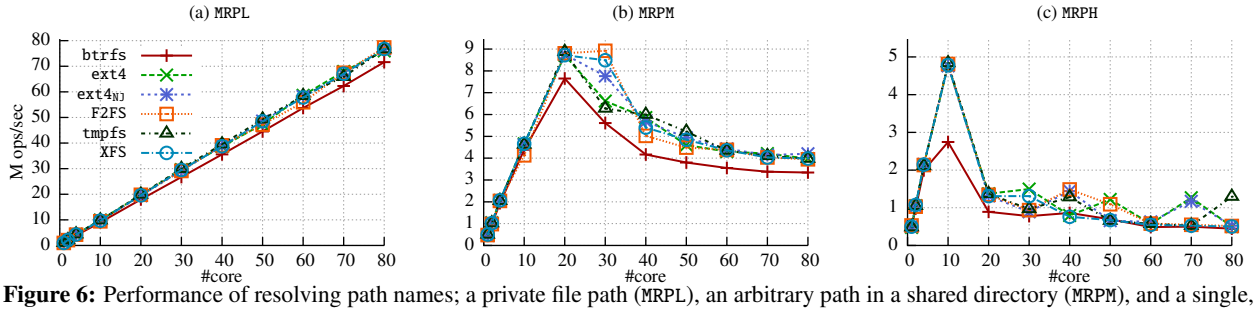


Figure 6: Performance of resolving path names; a private file path (MRPL), an arbitrary path in a shared directory (MRPM), and a single, shared path (MRPH). Surprisingly, resolving a single, common path name is the slowest operation (MRPH).

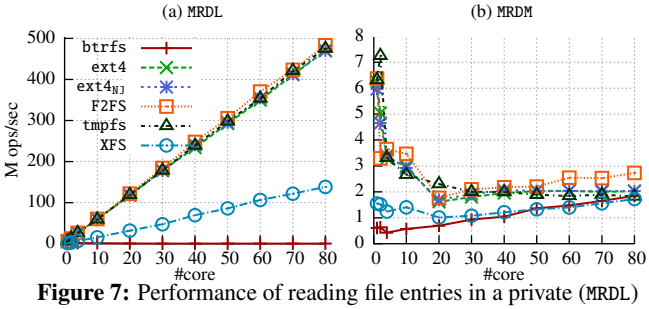


Figure 7: Performance of reading file entries in a private (MRDL) and a shared directory (MRDM). btrfs fails to scale in MRDL and none of the file systems scales in MRDM.

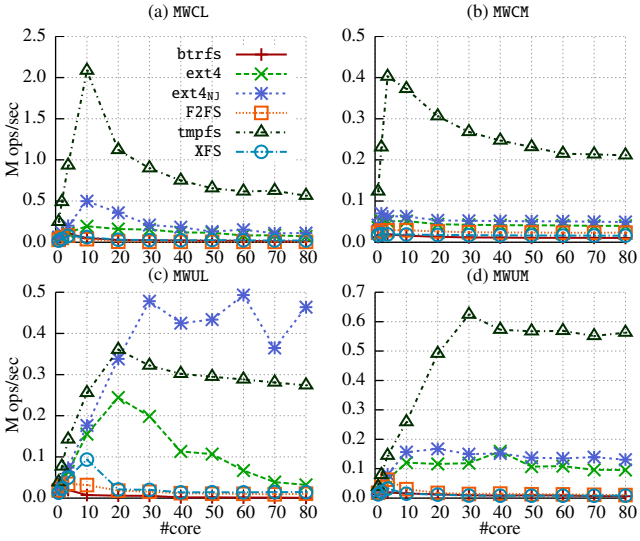


Figure 8: Performance of creating and deleting files in a private directory (i.e., MWCL and MWUL) and a shared directory (i.e., MWCM and MWUM).

5.2 Operation on File System Metadata

5.2.1 Path Name Resolution

The Linux kernel maintains a directory cache, or dcache, which caches dentry structures. Only when a dcache miss happens, the kernel calls the underlying file system to fill up the dcache. Since dcache hits are dominant in our microbenchmark, MRPL, MRPM, and MRPH stress the dcache operations implemented in the VFS layer. Thus there is little performance difference among file systems, as shown in Figure 6. Our experiment shows that (1) dcache is scalable up to 10-core if multiple processes attempt to resolve the occasionally shared path names (MRPM), and (2) contention on a shared path is so serious that resolving a single common path in applications becomes a scalability bottleneck. In MRPM and MRPH, a lockref in a dentry (i.e., dentry->d_lockref), which combines a spinlock and a reference count into a single locking primitive for better scalability [38], becomes a scalability bottleneck. Specifically, the CPI of ext4 in MRPH increases from 14.2 at 10-core to 20 at 20-core due to increased cache-coherence delays.

5.2.2 Directory Read

When listing a private directory (i.e., MRDL in Figure 7), all file systems scale linearly, except for btrfs. Ironically, the performance bottleneck of btrfs is the fine-grained locking. To read a file system buffer (i.e., extent_buffer) storing directory entries, btrfs first acquires read locks from a leaf, containing the buffer, to the root node of its B-tree (btrfs_set_lock_blocking_rw()); moreover, to acquire a read lock of a file system buffer, btrfs performs two read/write spinlock operations and six atomic operations for reference counting. Because such excessive synchronization operations increase cache-coherence delays, CPI in btrfs is 20.4× higher than that of ext4 at 80-core (20 and 0.98, respectively). XFS shows better scalability than btrfs even though its directory is represented as a B+-tree due to coarser-grained locking, i.e., per-directory locking.

Unexpectedly, listing the shared directory (i.e., MRDM in Figure 7) is not scalable in any of the file systems; the VFS holds an inode mutex before calling a file system-specific directory iteration function (iterate_dir()).

5.2.3 File Creation and Deletion

File creation and deletion performance are critical to the performance of email servers and file servers, which frequently create and delete small files. However, none of the file systems scale, as shown in Figure 8.

In tmpfs, a scalability bottleneck is adding and deleting a new inode in an inode list in a super block (i.e., sb->s_inodes). An inode list in a super block is protected by a system-wide (not file system-wide) spinlock (i.e., inode_sb_list_lock), so the spinlock becomes a performance bottleneck and incurs a performance col-

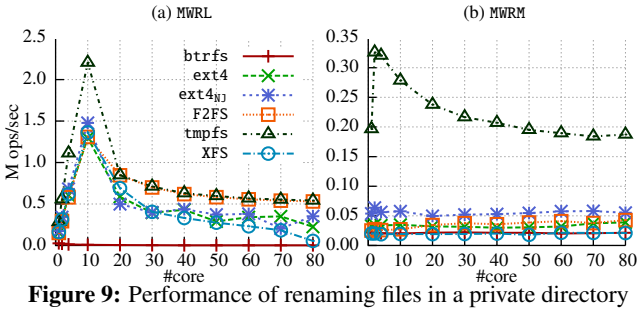


Figure 9: Performance of renaming files in a private directory (`MWRL`) and shared directory (`MWRM`). None of file systems scale.

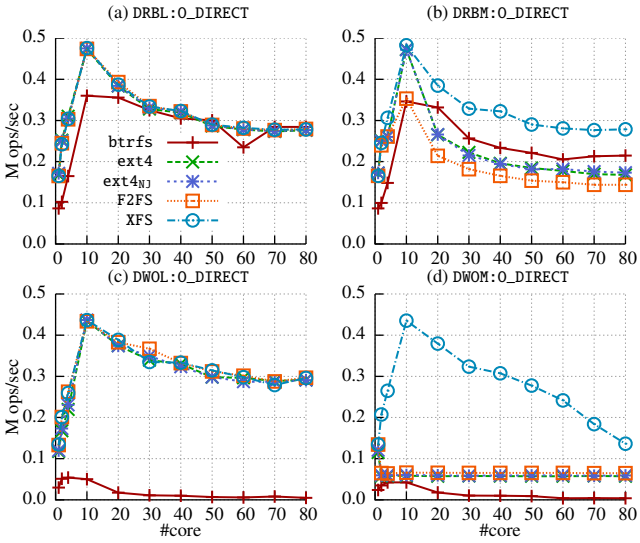


Figure 10: Performance of file data operations in a direct I/O mode on a single `RAMDISK`. Performance of all tested file systems are saturated or declining after 10-core, at which a single `RAMDISK` becomes a bottleneck. For write operations, `btrfs` suffers from its heavy b-tree operations the same as in the buffered mode. For `DWOM:O_DIRECT`, only `XFS` scales because it holds a shared lock for an `inode` while the others hold an inode mutex (`inode->i_mutex`).

lapse after 10-core. In fact, CPI at 10-core increases from 20 to 50 at 20-core due to shared cache line contention on the spinlock.

In `ext4`, inode allocation is a per-block group operation, so the maximum level of concurrency is the number of block groups (256 in our evaluation). But `ext4`'s policy to preserve spatial locality (e.g., putting files under the same directory to the same block group) limits the maximum concurrency. Upon file deletion, `ext4` first puts a deleted inode onto an orphan inode list in a super block (i.e., `sbi->s_orphan`), which is protected by a per-file-system spinlock. This list ensures that inodes and related resources (e.g., disk blocks) are freed even if the kernel crashes in the middle of the delete. Adding an inode to an orphan list is a sequential bottleneck.

Like `ext4`, `XFS` also maintains inodes per-block group (or allocation group). But, unlike `ext4`, a B+-tree is used to track which inode numbers are allocated and freed. Inode allocation and free incurs changes in the B+-tree and such changes need to be logged for consistency. So, journaling overhead waiting for flushing log buffers is the major source of bottlenecks (90% of time).

In `btrfs`, files and inodes are stored in the file system B-tree. Therefore, file creation and deletion incur changes in the file system B-tree, and such changes eventually need to be propagated to the root node. Similar to other write operations, updating the root node is again a sequential bottleneck. In fact, between 40% and 60% of execution time is spent contending to update the root node.

In `F2FS`, performance characteristics of file creation and deletion are similar to those of appending and truncating data blocks in §5.1.3. The reason for this, in the case of deletion, is the contention in updating the SIT (segment info table), which keeps track of blocks in active use. In fact, up to 85% of execution time is spent on contending for updating the SIT. During create, contention within the NAT (node address table) is the main reason for the performance collapse.

When creating and deleting files in a shared directory, additional contention updating the shared directory is noticeable (see `MWCM` and `MWUM` in Figure 8). Like `MRDM`, Linux `VFS` holds a per-directory mutex while creating and deleting files.

5.2.4 File Rename

As Figure 9 shows, rename is a system-wide sequential operation regardless of the sharing level. In `VFS`, multiple readers optimistically access the `dcache` through `rename_lock`, concurrently with multiple writers, and then later, each reader checks if the sequence number is the same as the one at the beginning of the operation. If sequence numbers do not match (i.e., there were changes in dentries), the reader simply retries the operation. Therefore, a rename operation needs to hold a write lock (i.e., `write_seqlock()` on `rename_lock`), which turns out to be the bottleneck in our benchmark. In `MWRL`, on average, 84.7% of execution time is spent waiting to acquire the `rename_lock`. This scalability bottleneck is a serious limitation for applications that concurrently perform renaming of multiple files, like Exim.

5.3 Scalability in a Direct I/O Mode

Performance in direct I/O mode is critical for many I/O-intensive applications. To understand the scalability behavior, we ran microbenchmarks on file data operations from §5.1 in direct I/O mode (`O_DIRECT`).

When each test process reads a block in its respective private file (`DRBL:O_DIRECT`), there is no apparent contention at file system so the storage device (i.e., a single `RAMDISK`) becomes the bottleneck. When more than 10 cores are used (i.e., more than two sockets are involved in our experimental setup), performance gradually degrades due to the NUMA effect. When reading a private block

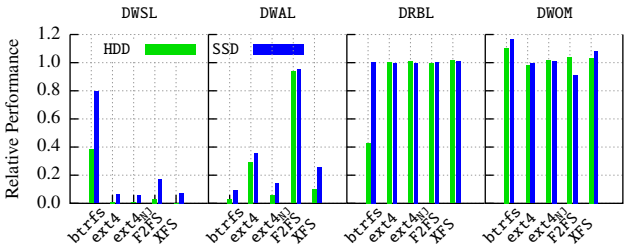


Figure 11: Relative performance of SSD and HDD to RAMDISK at 80-core. For buffered reads (e.g., DRBL) and contending operations (e.g., DWOM), the performance of the storage medium is not the dominant factor of applications' I/O performance.

of the shared file (DRBM:O_DIRECT), XFS shows around 20-50% higher performance than the other file systems. The performance difference comes from the different locking mechanism of the shared file for *w*riting. As discussed in §5.1.2, the file systems should lock the shared file before reading the disk blocks as the dirty pages of a file should be consistently written to that shared file. While writing dirty pages of a file in a direct I/O mode, XFS holds the shared lock of a file but others holds the inode mutex (inode->i_mutex). Thus, read operations of the other file systems are serialized by the inode mutex.

For write operations, btrfs suffers from its heavy b-tree operations regardless of the contention level. When writing private files (DWOL:O_DIRECT), the storage device is the bottleneck as same as DRBL:O_DIRECT. When writing a private block of the shared file (DWOM:O_DIRECT), only XFS scales up to 10-core. File systems other than XFS serialize concurrent write operations by holding the inode mutex. In contrast, since XFS holds the shared lock while writing disk blocks, write operations for the shared file can be concurrently issued.

The scalability bottleneck in accessing the shared file is a serious limitation for applications such as database systems and virtual machines, where large files (e.g., database table or virtual disk) are accessed in a direct I/O mode by multiple I/O threads.

5.4 Impact of Storage Medium

To understand how the performance of the storage medium affects the scalability behavior, we ran FxMark on SSD and HDD, and compared their performance at 80-core in Figure 11. For synchronous write operations (e.g., DWSL) or operations incurring frequent page cache misses (e.g., DWAL), the bandwidth of the storage medium is a dominant factor. However, for buffered reads (e.g., DRBL) or contending operations (e.g., DWOM), the impact of the storage medium is not dominant. With larger memory devices, faster storage mediums (e.g., NVMe), and increasing core counts in modern computers, it is important to understand, measure, and thus improve the scalability behavior of file systems.

6 Application Benchmarks Analysis

In this section, we explain the scalability behavior of three applications on various file systems backed by memory.

Exim. After removing the scalability bottleneck in Exim (see §3.2), it linearly scales up to 80-core (tmpfs in Figure 12(a)). With the optimized Exim, ext4 scales the most, followed by $\texttt{ext4}_{\texttt{NJ}}$, but it is still 10× slower than tmpfs. Since Exim creates and deletes small files in partitioned spool directories, performance bottlenecks in each file system are equivalent to both MWCL and MWUL (see §5.2.3).

RocksDB. As Figure 12(b) illustrates, RocksDB scales fairly well for all file systems up to 10 cores but either flattens out or collapses after that. The main bottleneck can be found in RocksDB itself, synchronizing compactor threads among each other. Since multiple compactor threads concurrently write new merged files to disk, the behavior and performance bottleneck in each file system is analogous to DWAL (see §5.1.2).

DBENCH. Figure 12(c) illustrates the DBENCH results, which do not scale linearly with increasing core count for any of the file systems. This happens because DBENCH reads, writes, and deletes a large number of files in a *shared directory*. This is similar to our microbenchmarks MWCM and MWUM (§5.1.3). tmpfs suffers for two reasons: look-ups and insertions in the page cache and reference counting for the dentry of the directory.

7 Summary of Benchmarks

We found 25 scalability bottlenecks in five widely-used file systems, as summarized in Figure 13 and Table 3. Some of the bottlenecks (e.g., inode list lock) are also found in recent literature [21, 56, 57]. In our opinion, this is the most important first step to scale file systems. We draw the following observations, to which I/O-intensive application developers must pay close attention.

High locality can cause performance collapse. Maintaining high locality is believed to be the golden rule to improve performance in I/O-intensive applications because it increases the cache hit ratio. But when the cache hit is dominant, the *scalability of cache hits* does matter. We found such performance collapses in the page cache and dentry cache in Linux file systems. [§5.1.1, §5.2.1]

Renaming is system-wide sequential. rename() is commonly used in many applications for transactional updates [46, 71, 75]. However, we found that rename() operations are completely serialized at a system level in Linux for consistent updates of the dentry cache. [§5.2.4]

Even read operations on a directory are sequential. All operations (e.g., readdir(), create(), and unlink()) on a shared directory are serialized through a per-directory mutex (inode->i_mutex) in Linux. [§5.2.2, §5.2.3, §5.2.4]

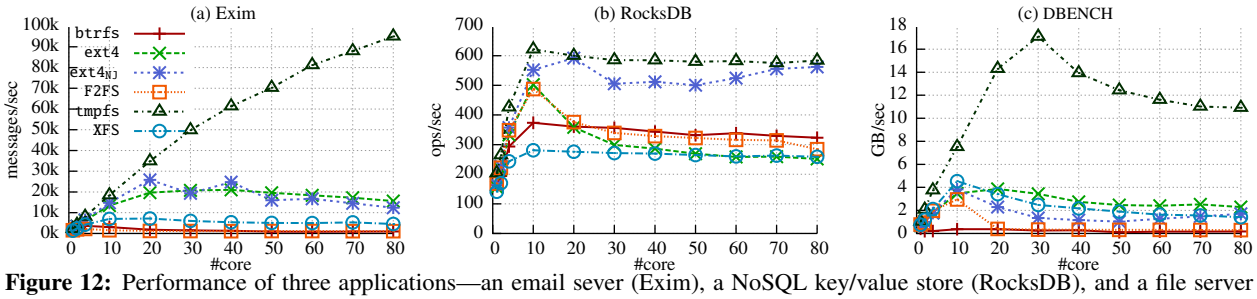


Figure 12: Performance of three applications—an email sever (Exim), a NoSQL key/value store (RocksDB), and a file server (DBENCH)—on various file systems backed by memory.

FS	Bottleneck	Sync. object	Scope	Operation
VFS	Rename lock	`rename_lock`	System	`rename()`
	`inode` list lock	`inode_sb_list_lock`	System	File creation and deletion
	Directory access lock	`inode->i_mutex`	Directory	All directory operations
	Page reference counter	`page->_count`	Page	Page cache access
	`dentry lockref` [38]	`dentry->d_lockref`	`dentry`	Path name resolution
`btrfs`	Acquiring a write lock for a B-tree node	`btrfs_tree_lock()`	File system	All write operations
	Acquiring a read lock for a B-tree node	`btrfs_set_lock_blocking_rw()`	File system	All read operations
	Checking data free space	`data_info->lock`	File system	File append
	Reserving data blocks	`delalloc_block_rsv->lock`	File system	File append
	File write lock	`inode->i_mutex`	File	File write
`ext4`	Acquiring a read lock for a journal	`journal->j_state_lock`	Journal	Heavy metadata update operations
	Atomic increment of a transaction counter	`t_handle_count`	Journal	Heavy metadata update operations
	Orphan `inode` list	`sbi->s_orphan`	File system	File deletion
	Block group lock	`bgl->locks`	Block group	File creation and deletion
	File write lock	`inode->i_mutex`	File	File write
`F2FS`	Single-threaded writing	`sbi->cp_rwsem`	File system	File write
	SIT (segment information table)	`sit_i->sentry_lock`	File system	File write
	NAT (node address table)	`nm_i->nat_tree_lock`	File system	File creation, deletion, and write
	File write lock	`inode->i_mutex`	File	File write
`tmpfs`	Cgroup page reference counter	`counter->count`	cgroup	File truncate
	Capacity limit check (per-CPU counter)	`sbinfo->used_blocks`	File system	File write near disk full
	File write lock	`inode->i_mutex`	File	File write
`XFS`	Journal writing	`log->l_icloglock`	File system	Heavy metadata update operations
	Acquiring a read lock of `XFS inode`	`ip->i_iolock`	File	File read
	File write lock	`inode->i_mutex`	File	File write

Table 3: The identified scalability bottlenecks in tested file systems with FxMark.

A file cannot be concurrently updated. All of the tested file systems hold an exclusive lock for a file (`inode->i_mutex`) during a write operation. This is a critical bottleneck for high-performance database systems allocating a large file but not maintaining the page cache by themselves (e.g., PostgreSQL). Even in the case of using direct I/O operations, this is the critical bottleneck for both read and write operations as we discussed in §5.3. To the best of our knowledge, the only way to concurrently update a file in Linux is to update it on a `XFS` partition with `O_DIRECT` mode, because `XFS` holds a shared lock of a file for all IO operations in direct I/O mode [26]. [§5.1.2]

Metadata changes are not scalable. For example, creating, deleting, growing, and shrinking a file are not scalable in Linux. The key reason is that existing consistency mechanisms (i.e., journaling in `ext4` and `XFS`, copy-on-write in `btrfs`, and log-structured writing in `F2FS`) are not designed with manycore scalability in mind. [§5.1.3, §5.2.3]

Overwriting could be as expensive as appending. Many I/O-optimized applications such as MySQL and PostgreSQL pre-allocate file blocks of heavily updated files (e.g., log files) and overwrite them instead of growing files dynamically to reduce the overhead of changing its metadata [64, 71, 84]. But in non-in-place update file systems such as `btrfs` and `F2FS`, overwrite is written in a new place; it involves freeing and allocating disk blocks, updating the inode block map, etc. and thus is as expensive as append operations. [§5.1.2]

Scalability is not portable. Some file systems have peculiar scalability characteristics for some operations; a single file read of multiple threads is not scalable in `XFS` due to the scalability limitation of Linux's read/write semaphore; in `F2FS`, a checkpointing operation caused by segment cleaning freezes entire file system operations so scalability will be seriously hampered under write-heavy workloads with low free space; enumerating directories in `btrfs` is not scalable because of too frequent atomic operations. The scalability of an I/O-intensive application is very file system-specific. [§5.1.1, §5.1.2, §5.2.2]

Non-scalability often means wasting CPU cycles. In file systems, concurrent operations are coordinated mainly by locks. Because of spinning of spinlock and optimistic spinning of blocking locks, non-scalable file systems tend

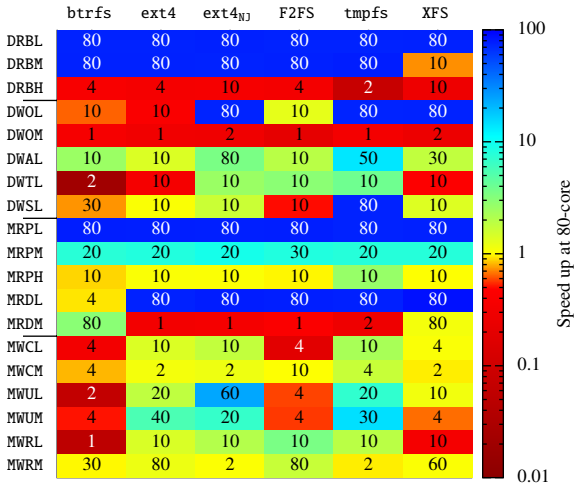


Figure 13: Summary of manycore scalability of tested file systems. The color represents the relative speed over a single-core performance of a file system with a specific microbenchmark. The number in a cell denotes the core count at peak performance.

to consume more CPU cycles to alleviate the contention. In our benchmarks, about 30-90% of CPU cycles are consumed for synchronization. [§5.1.3, §5.2.2]

8 Discussion

The next question is whether traditional file system designs can be used and implemented in a scalable way. It is difficult to answer conclusively. At a high level, the root causes of the scalability problems of file systems are not different from those of OS scalability in previous studies [21, 22, 32–34]: shared cache line contention, reference counting, coarse-grained locking, etc. But the devil is in the details; some are difficult to fix with known techniques and lead us to revisit the core design of file systems.

Consistency mechanisms need to be scalable. All three consistency mechanisms, journaling (`ext4` and `XFS`), log-structured writing (`F2FS`), and copy-on-write (`btrfs`), are scalability bottlenecks. We think these are caused by their inherent designs so that our community needs to revisit consistency mechanisms from a scalability perspective.

In journaling file systems, committing a journal transaction guarantees a filesystem-wide consistency. To this end, `ext4`, for example, maintains a single running journal transaction, so accessing the journal transaction becomes a scalability bottleneck. Scalable journaling is still an unexplored area in the context of file systems, though there are some studies in the database field [54, 66, 90, 91].

In the case when copy-on-write techniques are combined with self-balancing index structures (e.g., B-tree) like `btrfs`, such file systems are very fragile to scalability; a leaf node update triggers updates of all interim nodes to the root so that write locks of all nodes should be acquired. Moreover, two independent updates should contend for acquiring locks of common ancestors. Besides locking overhead, this could result in a deadlock if two updates should be coordinated by other locks. We suspect this is the reason why `btrfs` uses the retry-based locking protocol to acquire a node lock (`btrfs_tree_lock()`). Parallelizing a CoW file system by extending the current B-tree scheme (e.g., LSM tree) or using non-self-balancing index structures (e.g., radix tree or hash table) is worth further research.

To our best knowledge, all log-structured file systems, including `F2FS`, `NILFS2` [3], and `UBIFS` [4], adopt single-threaded writing. By nature, log-structured file systems are designed to create a large sequential write stream, while metadata updates should be issued after writing file data for consistency guarantee. Multi-headed log-structured writing schemes are an unexplored area in the context of file systems, while some techniques are proposed at the storage device level [28, 58].

Spatial locality still needs to be considered. One potential solution to parallelizing file systems is partitioning. To see its feasibility, we modified Exim and RocksDB to run on multiple file system partitions for spool directories and database files, respectively. We set up 60 file system partitions, the maximum allowed on a disk, to spread files in 60 ways. Our results on `RAMDISK` and `HDD` show its potential and limitations at the same time. We see a significant performance improvement on `RAMDISK` (Figure 14). It confirms that the reduced contentions in a file system can improve the scalability. However, the RocksDB results on `HDD` also show its limitation (Figure 15). In all file systems except for `F2FS`, the partitioned case performs worse, as partitioning ruins spatial locality. But `F2FS` performs equally well in both cases; because the log-structured writing scheme of `F2FS` always issues bulk sequential write for file data, the impact of partitioning is negligible. The above example shows the unique challenges in designing scalable file systems. Optimizing for storage devices based on their performance characteristics and achieving consistency guarantees in a scalable fashion will be critical in file systems.

File system-specific locking schemes in VFS. The scalable performance of locking strategies, such as granularity and types of lock, is dependent on data organization and management. The current locking schemes enforced in `VFS` will become obsolete as storage devices and file systems change. For example, the inode mutex for directory accesses, currently enforced by the `VFS`, should be flexible enough for each file system to choose proper, finer-grained locking.

Reference counting still does matter. We found scalability problems in the reference counters of various file system objects (e.g., `page`, `dentry`, and `XFS inode` struc-

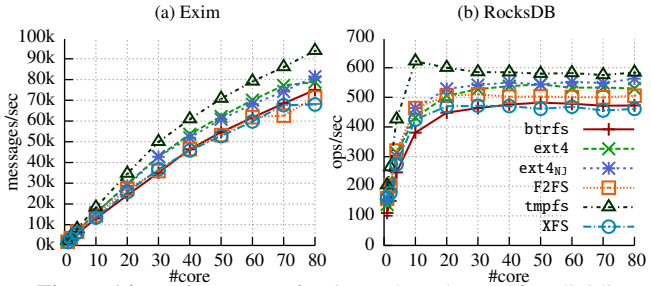


Figure 14: Performance of Exim and RocksDB after dividing each file system to 60 partitions on `RAMDISK`.

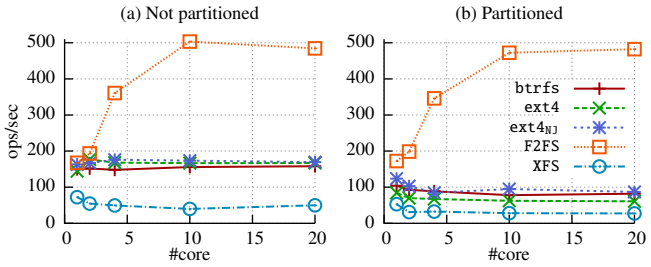


Figure 15: Performance of RocksDB with and without partitioning `HDD`. For clarity, results up to 20-core are presented because the performance of all file systems is saturated after that. Except for `F2FS`, all other file systems fail to scale after partitioning and perform around 50% of the original performance. The impact of partitioning in `F2FS` is negligible because the log-structured writing in `F2FS` always generates large sequential write for file data.

tures). Many previous studies proposed scalable reference counting mechanisms, including clustered object [61], SNIZ [40], sloppy counter [21], Refcache [33], and Linux per-CPU counter [39], but we identified that they are not adequate for file system objects for two reasons. First, their space overhead is proportional to the number of cores since they speed up by separating cache lines per-core. Such space overhead is especially problematic to file system objects, which are many in number and small in size (typically a few tens or hundreds of bytes). For example, in our 80-core machine, the space required for per-core counters per `page` is 5× larger than the `page` structure itself (320 bytes vs. 64 bytes). Second, getting the true value is not scalable but immediate. Recall that we found the `reader` counter problem at the R/W semaphore used in the `XFS` `inode` (§5.1.1). If getting the number of `reader` is not immediate, writers waiting for readers can starve.

9 Related Work

Benchmarking file systems. Due to the complexity of file systems and interactions among multiple factors (e.g., page cache, on-disk fragmentation, and device characteristics), file system benchmarks have been criticized for decades [12, 87–89]. Popular file system benchmarks, such as FIO [1] and iozone [2], mostly focus on measuring bandwidth and IOPS of file system operations varying I/O patterns. In contrast, recently developed benchmarks focus on a specific system (e.g., smartphones [59]) or a particular component in file systems (e.g., block allocation [47]). Along this line, FxMark focuses only on manycore scalability of file systems.

Scaling operating systems. To improve the scalability of OS, researchers have been optimizing existing OSes [20–22, 32, 33] or have been building new OSes based on new design principles [16, 34]. However, previous studies used memory-based file systems to opt out of the effect of I/O operations. In Arrakis [74], since file system service is a part of applications, its manycore scalability solely depends on each application.

Scaling file systems. The Linux kernel community has made a steady effort to improve the scalability of the file system by mostly reducing lock contentions [35, 36, 65]. Hare [45] is a scalable file system for non-cache-coherent systems. But it does not provide durability and crash consistency, which were significant performance bottlenecks in our evaluation. ScaleFS [41] extends a scalable in-memory file system to support consistency by using operation log on an on-disk file system. SpanFS [57] adopts partitioning techniques to reduce lock contentions. But how partitioning affects performance in a physical storage medium such as SSD and HDD is not explored.

Optimizing the storage stack for fast storage. As storage devices become dramatically faster, there are research efforts to make storage stacks more scalable. Many researchers made efforts to reduce the overhead and latency of interrupt handling in the storage device driver layer [13, 82, 92, 93]. At the block layer, Bjørling et al. [18] address the scalability of the Linux block layer and propose a new Linux block layer, which maintains a per-core request queue.

10 Conclusion

We performed a comprehensive analysis of the manycore scalability of five widely-deployed file systems using our FxMark benchmark suite. We observed many unexpected scalability behaviors of file systems. Some of them lead us to revisit the core design of traditional file systems; in addition to well-known scalability techniques, scalable consistency guarantee mechanisms and optimizing for storage devices based on their performance characteristics will be critical. We believe that our analysis results and insights can be a starting point toward designing scalable file systems for manycore systems.

11 Acknowledgment

We thank the anonymous reviewers, and our shepherd, Angela Demke Brown, for their helpful feedback. This research was supported by the NSF award DGE-1500084, ONR under grant N000141512162, DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983].

References

[1] Flexible I/O Tester. https://github.com/axboe/fio.

[2] IOzone Filesystem Benchmark. http://www.iozone.org/.

[3] NILFS – Continuous Snapshotting Filesystem. http://nilfs.sourceforge.net/en/.

[4] UBIFS – UBI File-System. http://www.linux-mtd.infradead.org/doc/ubifs.html.

[5] DBENCH, 2008. https://dbench.samba.org/.

[6] MongoDB, 2009. https://www.mongodb.org/.

[7] perf: Linux profiling with performance counters , 2014. https://perf.wiki.kernel.org/index.php/Main_Page.

[8] Exim Internet Mailer, 2015. http://www.exim.org/.

[9] SAP HANA, 2015. http://hana.sap.com/abouthana.html/.

[10] MariaDB, 2015. https://mariadb.org/.

[11] VoltDB, 2015. https://voltdb.com/.

[12] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating Realistic Impressions for File-system Benchmarking. *Trans. Storage*, 5(4):16:1–16:30, Dec. 2009.

[13] I. Ahmad, A. Gulati, and A. Mashtizadeh. vIC: Interrupt coalescing for virtual machine storage device IO. In *Proceedings of the 2011 ATC Annual Technical Conference (ATC)*, Portland, OR, June 2011.

[14] P. Alcorn. Samsung Releases New 12 Gb/s SAS, M.2, AIC And 2.5" NVMe SSDs: 1 Million IOPS, Up To 15.63 TB, 2013. http://www.tomsitpro.com/articles/samsung-sm953-pm1725-pm1633-pm1633a,1-2805.html.

[15] Artem B. Bityutskiy. JFFS3 design issues. http://linux-mtd.infradead.org/tech/JFFS3design.pdf.

[16] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[17] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*, Ottawa, Canada, June 2003.

[18] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, June 2013.

[19] S. Borkar. Thousand Core Chips: A Technology Perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC)*, June 2007.

[20] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many Cores. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[21] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[22] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2012.

[23] N. Brown. An f2fs teardown, 2010. https://lwn.net/Articles/518988/.

[24] M. Callaghan. Bug #55004: async fuzzy checkpoint constraint isn't really async, 2010. http://bugs.mysql.com/bug.php?id=55004.

[25] M. Callaghan. InnoDB fuzzy checkpoints, 2010. https://www.facebook.com/notes/mysqlfacebook/innodb-fuzzy-checkpoints/408059000932.

[26] M. Callaghan. XFS, ext and per-inode mutexes, 2011. https://www.facebook.com/notes/mysql-at-facebook/xfs-ext-and-per-inode-mutexes/10150210901610933.

[27] M. Cao, J. R. Santos, and A. Dilger. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium*, 2008.

[28] M.-L. Chiang, P. C. Lee, and R.-C. Chang. Using data clustering to improve cleaning performance for flash memory. *Software-Practice & Experience*, 29(3):267–290, 1999.

[29] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2012.

[30] D. Chinner. XFS Delayed Logging Design, 2010. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/xfs-delayed-logging-design.txt.

[31] D. Chinner. Improving Metadata Performance By Reducing Journal Overhead, 2010. http://xfs.org/index.php/Improving_Metadata_Performance_By_Reducing_Journal_Overhead.

[32] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, Mar. 2012.

[33] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the ACM EuroSys Conference*, Prague, Czech Republic, Apr. 2013.

[34] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[35] J. Corbet. JLS: Increasing VFS scalability, 2009. https://lwn.net/Articles/360199/.

[36] J. Corbet. Dcache scalability and RCU-walk, 2010. https://lwn.net/Articles/419811/.

[37] J. Corbet. XFS: the filesystem of the future?, 2012. https://lwn.net/Articles/476263/.

[38] J. Corbet. Introducing lockrefs, 2013. https://lwn.net/Articles/565734/.

[39] J. Corbet. Per-CPU reference counts, 2013. https://lwn.net/Articles/557478/.

[40] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable NonZero Indicators. In *Proceedings of the 26th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[41] R. Eqbal. ScaleFS: A Multicore-Scalable File System. Master's thesis, Massachusetts Institute of Technology, 2014.

[42] Ext4 Wiki. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[43] Facebook. RocksDB, 2013. http://rocksdb.org/.

[44] M. J. Foley. Microsoft SQL Server 2014 released to manufacturing, 2014. http://www.zdnet.com/article/microsoft-sql-server-2014-released-to-manufacturing/.

[45] C. Gruenwald III, F. Sironi, M. F. Kaashoek, and N. Zeldovich.

Hare: a file system for non-cache-coherent multicores. In *Proceedings of the ACM EuroSys Conference*, Bordeaux, France, Apr. 2015.

[46] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[47] J. He, D. Nguyen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Reducing file system tail latencies with Chopper. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015.

[48] HP. Performance improvements using Concurrent I/O on HP-UX 11i v3 with OnlineJFS 5.0.1 and the HP-UX 11i Logical Volume Manager, 2015. http://www.filibeto.org/unix/hp-ux/lib/os/volume-manager/perf-hpux-11.31-cio-onlinejfs-4AA1-5719ENW.pdf.

[49] *Converged System for SAP HANA Scale-out Configurations*. HP, 2015. http://www8.hp.com/h20195/v2/GetPDF.aspx%2F4AA5-1488ENN.pdf.

[50] IBM. Use concurrent I/O to improve DB2 database performance, 2012. http://www.ibm.com/developerworks/data/library/techarticle/dm-1204concurrent/.

[51] Intel. Performance Benchmarking for PCIe and NVMe Enterprise Solid-State Drives, 2015. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/performance-pcie-nvme-enterprise-ssds-white-paper.pdf.

[52] D. Jeong, Y. Lee, and J.-S. Kim. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015.

[53] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35. ACM, 2009.

[54] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.*, 3(1-2):681–692, Sept. 2010.

[55] D. H. Kang, C. Min, and Y. I. Eom. An Efficient Buffer Replacement Algorithm for NAND Flash Storage Devices. In *Proceedings of the 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, Paris, France, Sept. 2014.

[56] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai. MultiLanes: providing virtualized storage for OS-level virtualization on many cores. In *Proceedings of the 12th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2014.

[57] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. SpanFS: a scalable file system on fast storage devices. In *Proceedings of the 2015 ATC Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[58] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, pages 13–13. USENIX Association, 2014.

[59] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting Storage for Smartphones. *Trans. Storage*, 8(4):14:1–14:25, Dec. 2012.

[60] J. Kim. f2fs: introduce flash-friendly file system , 2012. https://lwn.net/Articles/518718/.

[61] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig. K42: Building a Complete Operating System. In *Proceedings of the ACM EuroSys Conference*, Leuven, Belgium, Apr. 2006.

[62] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. ATAC: A 1000-core Cache-coherent Processor with On-chip Optical Network. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vienna, Austria, Sept. 2010.

[63] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th Usenix Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, Feb. 2015.

[64] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *Proceedings of the 2015 ATC Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[65] W. Long. [PATCH] dcache: Translating dentry into pathname without taking rename_lock, 2013. https://lkml.org/lkml/2013/9/4/471.

[66] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshop*, Chicago, IL, Mar. 2014.

[67] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium*, 2007.

[68] R. McDougall. Solaris Internals and Performance FAQ: Direct I/O, 2012. http://www.solarisinternals.com/wiki/index.php/Direct_I/O.

[69] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, Aug. 1984. ISSN 0734-2071.

[70] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2012.

[71] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the 2015 ATC Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[72] T. P. Morgan. Flashtec NVRAM Does 15 Million IOPS At Sub-Microsecond Latency, 2014. http://www.enterprisetech.com/2014/08/06/flashtec-nvram-15-million-iops-sub-microsecond-latency/.

[73] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[74] S. Peter, J. Li, I. Zhang, D. R. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[75] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[76] A. Raj. CPU hotplug Support in Linux(tm) Kernel, 2006. https://www.kernel.org/doc/Documentation/cpu-hotplug.txt.

[77] O. Rodeh. B-trees, Shadowing, and Clones. *Trans. Storage*, 3(4):

2:1–2:27, Feb. 2008. ISSN 1553-3077.

[78] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[79] C. Rohland. Tmpfs is a file system which keeps all files in virtual memory, 2001. git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/tmpfs.txt.

[80] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992. ISSN 0734-2071.

[81] F. B. Schmuck and R. L. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st Usenix Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[82] D. I. Shin, Y. J. Yu, H. S. Kim, J. W. Choi, D. Y. Jung, and H. Y. Yeom. Dynamic interval polling and pipelined post i/o processing for low-latency storage class memory. In *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*, pages 5–5. USENIX Association, 2013.

[83] Silicon Graphics Inc. XFS Filesystem Structure, 2006. http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure/tmp/en-US/html/index.html.

[84] J. Swanhart. An Introduction to InnoDB Internals, 2011. https://www.percona.com/files/percona-live/justin-innodb-internals.pdf.

[85] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the 1996 ATC Annual Technical Conference (ATC)*, Jan. 1996.

[86] B. Tallis. Intel Announces SSD DC P3608 Series, 2015. http://www.anandtech.com/show/9646/intel-announces-ssd-dc-p3608-series.

[87] D. Tang and M. Seltzer. Lies, damned lies, and file system benchmarks. Technical report, Technical Report TR-34-94, Harvard University, 1994.

[88] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It* is* rocket science. *HotOS XIII*, 2011.

[89] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright. A Nine Year Study of File System and Storage Benchmarking. *Trans. Storage*, 4(2):5:1–5:56, May 2008.

[90] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, Nov. 2013.

[91] T. Wang and R. Johnson. Scalable Logging Through Emerging Non-volatile Memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[92] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th Usenix Conference on File and Storage Technologies (FAST)*, San Jose, California, USA, Feb. 2012.

[93] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, H. Eom, and H. Y. Yeom. Exploiting peak device throughput from random access workload. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*. USENIX Association, 2012.

ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices

Jiacheng Zhang Jiwu Shu* Youyou Lu
Department of Computer Science and Technology, Tsinghua University
Tsinghua National Laboratory for Information Science and Technology
zhang-jc13@mails.tsinghua.edu.cn {*shujw, luyouyou*}*@tsinghua.edu.cn*

Abstract

File system designs are undergoing rapid evolution to exploit the potentials of flash memory. However, the internal parallelism, a key feature of flash devices, is hard to be leveraged in the file system level, due to the semantic gap caused by the flash translation layer (FTL). We observe that even flash-optimized file systems have serious garbage collection problems, which lead to significant performance degradation, for write-intensive workloads on multi-channel flash devices.

In this paper, we propose ParaFS to exploit the internal parallelism while ensuring efficient garbage collection. ParaFS is a log-structured file system over a simplified block-level FTL that exposes the physical layout. With the knowledge of device information, ParaFS first proposes 2-D data allocation, to maintain the hot/cold data grouping in flash memory while exploiting channel-level parallelism. ParaFS then coordinates the garbage collection in both FS and FTL levels, to make garbage collection more efficient. In addition, ParaFS schedules read/write/erase requests over multiple channels to achieve consistent performance. Evaluations show that ParaFS effectively improves system performance for write-intensive workloads by 1.6× to 3.1×, compared to the flash-optimized F2FS file system.

1 Introduction

Flash memory has been widely adopted across embedded systems to data centers in the past few years. In the device level, flash devices outperform hard disk drives (HDDs) by orders of magnitude in terms of both latency and bandwidth. In the system level, the latency benefit has been made visible to the software by redesigning the I/O stack [8, 43, 22, 11]. But unfortunately, the bandwidth benefit, which is mostly contributed by the internal parallelism of flash devices [13, 18, 10, 15], is underutilized in system softwares.

Researchers have made great efforts in designing a file system for flash storage. New file system architectures have been proposed, to either improve data allocation performance, by removing redundant functions between FS and FTL [20, 45], or improve flash memory endurance, by using an object-based FTL to facilitate hardware/software co-designs [33]. Features of flash memory have also been leveraged to redesign efficient metadata mechanisms. For instance, the imbalanced read/write feature has been studied to design a persistence-efficient file system directory tree [32]. F2FS, a less aggressive design than the above-mentioned designs, is a flash-optimized file system and has been merged into the Linux kernel [28]. However, the internal parallelism has not been well studied in these file systems.

Unfortunately, internal parallelism is a key design issue to improve file system performance for flash storage. Even though it has been well studied in the FTL level, file systems cannot always gain the benefits. We observe that, even though the flash-optimized F2FS file system outperforms the legacy file system Ext4 when write traffic is light, its performance does not scale well with the internal parallelism when write traffic is heavy (48% of Ext4 in the worst case, more details in Section 4.3 and Figure 6).

After investigating into file systems, we have the following three observations. First, optimizations in both FS and FTL are made but may collide. For example, data pages[1] are grouped into hot/cold groups in some file systems, so as to reduce garbage collection (GC) overhead. FTL stripes data pages over different parallel units (e.g., channels) to achieve parallel performance, but breaks up the hot/cold groups. Second, duplicated log-structured data management in both FS and FTL leads to inefficient garbage collection. FS-level garbage collection is unable

*Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).

[1] In this paper, we use *data pages* instead of *data blocks*, in order not to be confused with *flash blocks*.

to erase physical flash blocks, while FTL-level garbage collection is unable to select the right victim blocks due to the lack of semantics. Third, isolated I/O scheduling in either FS or FTL results in unpredictable I/O latencies, which is a new challenge in storage systems on low-latency flash devices.

To exploit the internal parallelism while keeping low garbage collection overhead, we propose ParaFS, a log-structured file system over simplified block-level FTL. The simplified FTL exposes the device information to the file system. With the knowledge of the physical layout, ParaFS takes the following three approaches to exploit the internal parallelism. First, it fully exploits the channel-level parallelism of flash memory by page-unit striping, while ensuring data grouping physically. Second, it coordinates the garbage collection processes in the FS and FTL levels, to make GC more efficient. Third, it optimizes the request scheduling on read, write and erase requests, and gains more consistent performance. Our major contributions are summarized as follows:

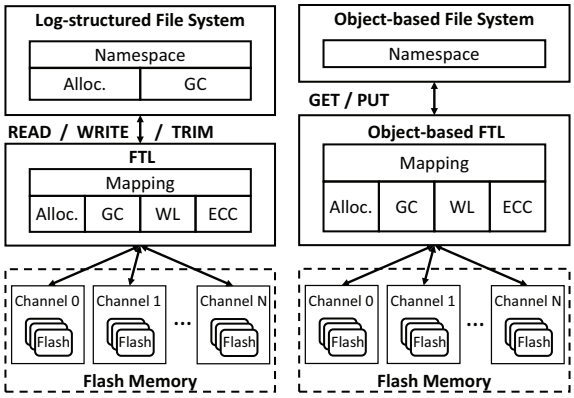
- We observe the internal parallelism of flash devices is underutilized when write traffic is heavy, and propose ParaFS, a parallelism-aware file system over a simplified FTL.
- We propose parallelism-aware mechanisms in the ParaFS file system, to mitigate the parallelism's conflicts with hot/cold grouping, garbage collection, and I/O performance consistency.
- We implement ParaFS in the Linux kernel. Evaluations using various workloads show that ParaFS has higher and more consistent performance with significant lower garbage collection overhead, compared to legacy file systems including the flash-optimized F2FS.

The rest of this paper is organized as follows. Section 2 analyses the parallelism challenges and motivates the ParaFS design. Section 3 describes the ParaFS design, including the 2-D allocation, coordinated garbage collection and parallelism-aware scheduling mechanisms. Evaluations of ParaFS are shown in Section 4. Related work is given in Section 5 and the conclusion is made in Section 6.

2 Motivation

2.1 Flash File System Architectures

Flash devices are seamlessly integrated into legacy storage systems by introducing a flash translation layer (FTL). Though NAND flash has low access latency and high I/O bandwidth, it has several limitations, e.g., erase-before-write and write endurance (i.e., limited program/erase cycles). Log-structured design is supposed to be a friendly way for flash memory. It is adopted in both the file system level (e.g., F2FS [28], and NILFS [27]) and the FTL level [10, 36]. The FTL hides the flash limitations by updating data in a log-structured way using an address mapping table. It exports the same I/O interface as HDDs. With the use of the FTL, legacy file systems can directly run on these flash devices, without any changes. This architecture is shown in Figure 1(a).



(a) Block Interface (b) Object Interface

Figure 1: FS Architectures on Flash Storage

While FTL is good at abstracting the flash memory as a block device, it widens the semantic gap between file systems and flash memory. As shown in Figure 1(a), the FS and FTL run independently without knowing each other's behaviours, resulting in inefficient storage management. Even worse, the two levels have redundant functions, e.g., space allocation, address mapping, and garbage collection, which further induce performance and endurance overhead.

Object-based FTL (OFTL) [33] is a recently proposed architecture to bridge the semantic gap between file systems and FTLs, as shown in Figure 1(b). However, it requires dramatic changes of current I/O stack, and is difficult to be adopted currently, either in the device due to complexity or in the operating system due to rich interfaces required in the drivers.

2.2 Challenges of Internal Parallelism

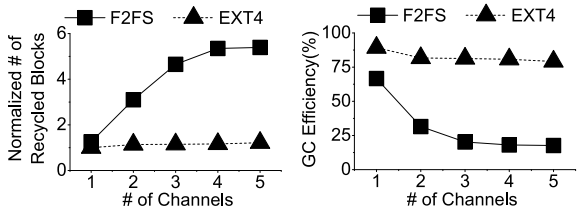
In a flash device, a number of flash chips are connected to the NAND flash controller through multiple channels. I/O requests are distributed to different channels to achieve high parallel performance, and this is known as the *internal parallelism*. Unfortunately, this feature has not been well leveraged in the file systems for the following three challenges.

Hot/Cold Grouping vs. Internal Parallelism. With the use of the FTL, file systems allocate data pages in a one-dimension linear space. The linear space works fine for HDDs, because a hard disk accesses sectors serially due to the mechanical structure, but not for flash devices.

In flash-based storage systems, a fine hot/cold grouping reduces the number of valid pages in victim flash blocks during garbage collection. Some file systems separate data into groups with different hotness for GC efficiency. Meanwhile, the FTL trys to allocate flash pages from different parallel units, aiming at high bandwidth. Data pages that belong to the same group in the file system may be written to different parallel units. Parallel space allocation in FTL breaks hot/cold data groups. As such, the hot/cold data grouping in the file system collides with the internal parallelism in the FTL.

Garbage Collection vs. Internal Parallelism. Log-structured file systems are considered to be flash friendly. However, we observe that F2FS, a flash-optimized log-structured file system, performs worse than Ext4 on multi-channel flash devices when write traffic is heavy. To further analyse this inefficiency, we implement a page-level FTL and collect the GC statistics in the FTL, including the number of erased flash blocks and the percentage of invalid pages in each erased block (i.e., GC efficiency). Details of the page-level FTL and evaluation settings are introduced in Section 4. As shown in Figure 2, for the random update workload in YCSB[14], F2FS has increasing number of erased flash blocks and decreasing GC efficiency when the number of channels increases.

The reason lies in the incoordinate garbage collections in FS and FTL. When the log-structured file system cleans a segment in the FS level, the invalid flash pages are actually scattered over multiple flash parallel units. The more channels the device has, the more diverse the invalid pages are. This degrades the GC efficiency in the FTL. At the same time, the FTL gets pages' invalidation only after receiving the *trim* commands from the log-structured file system, due to the no in-place update. As we observed in the experiments, a large number of pages which are already invalidated in the file system are migrated to the clean flash blocks during the FTL's erase process. Therefore, garbage collections in FS and FTL levels collide and damage performance.



(a) # of Recycled Blocks (b) GC Efficiency

Figure 2: GC Statistics under Heavy Traffic

Consistent Performance vs. Internal Parallelism. Read or write requests may be blocked by erase operations in the FTL. Since erase latency is much higher than read/write latency, this causes latency spikes, making I/O latency unpredictable. It is known as the inconsistent performance, which is a serious issue for low-latency storage devices. Parallel units offer an opportunity to remove latency spikes by carefully scheduling I/O requests. Once a file system controls read, write and erase operations, it can dynamically schedule requests to make performance more consistent.

To address the above-mentioned challenges, we propose a new architecture (as shown in Figure 3) to exploit the internal parallelism in the file system. In this architecture, we use a simplified FTL to expose the physical layout to the file system. Our designed ParaFS maximizes the parallel performance in the file system level while achieving physical hot/cold data grouping, lower garbage collection overhead and more consistent performance.

3 Design

ParaFS is designed to exploit the internal parallelism without compromising other mechanisms. To achieve this goal, ParaFS uses three key techniques:

- *2-Dimension Data Allocation* to stripe data pages over multiple flash channels at page granularity while maintaining hot/cold data separation physically.
- *Coordinated Garbage Collection* to coordinate garbage collections in FS and FTL levels and lower the garbage collection overhead.
- *Parallelism-Aware Scheduling* to schedule the read/write/erase requests to multiple flash channels for more consistent system performance.

In this section, we introduce the ParaFS architecture first, followed by the description of the above-mentioned three techniques.

3.1 The ParaFS Architecture

ParaFS reorganizes the functionalities between the file system and the FTL, as shown in Figure 3. In the ParaFS architecture, ParaFS relies on a simplified FTL (annotated as S-FTL). S-FTL is different from traditional FTLs in three aspects. First, S-FTL uses a static block-level mapping table. Second, it performs garbage collection by simply erasing the victim flash blocks without moving any pages, while the valid pages in the victim blocks are migrated by ParaFS in the FS level. Third, S-FTL exposes the physical layout to the file system using three values, i.e., the number of flash channels[2], the flash page size and the flash block size. These three values are passed to the ParaFS file system through *ioctl* interface,

[2]Currently, ParaFS exploits only the channel-level parallelism, while finer level (e.g., die-level, plane-level) parallelism can be exploited in the FTL by emulating large pages or blocks.

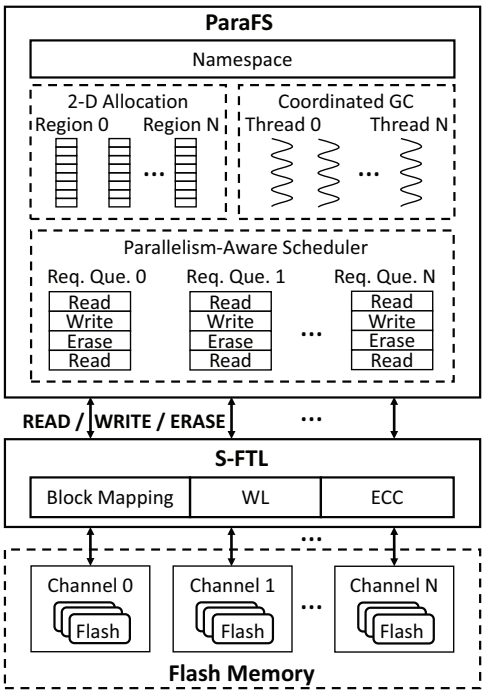


Figure 3: The ParaFS Architecture

when the device is formatted. With the physical layout information, ParaFS manages data allocation, garbage collection, and read/write/erase scheduling in the file system level. The ParaFS file system uses an I/O interface that consists of read/write/erase commands, which remains the same as the legacy block I/O interface. In ParaFS, the erase command reuses the *trim* command in the legacy protocol.

In S-FTL, the block mapping is different from those in legacy block-based FTLs. S-FTL uses static block-level mapping. For normal writes, the ParaFS file system updates data pages in a log-structured way. There is no in-place update in the FS level. For these writes, S-FTL doesn't need to remap the block. S-FTL updates its mapping entries only for wear leveling or bad block remapping. In addition, ParaFS tracks the valid/invalid statuses of pages in each segment, whose address is aligned to the flash block in flash memory. The file system migrates valid pages in the victim segments during garbage collection, and the S-FTL only needs to erase the corresponding flash blocks afterwards. As such, the simplified block mapping, simplified garbage collection and reduced functionalities make S-FTL a lightweight implementation, which has nearly zero overhead.

The ParaFS file system is a log-structured file system. To avoid the "wandering tree" problem [38], file system metadata is updated in place, by introducing a small page-level FTL. Since we focus on the data management, we omit discussions of this page-level FTL for the rest of the paper. In contrast, file system data is updated in a log-structured way and sent to the S-FTL. The ParaFS file system is able to perform more effective data management with the knowledge of the internal physical layout of flash devices. Since the file system is aware of the channel layout, it allocates space in different channels to exploit the internal parallelism and meanwhile keeps the hot/cold data separation physically (details in Section 3.2). ParaFS aligns the segment of log-structured file system to the flash block of the device. Since it knows exactly the valid/invalid statuses of data pages, it performs garbage collection in the file system with the coordination of the FTL erase operations (details in Section 3.3). With the channel information, ParaFS is also able to optimize the scheduling on read/write/erase requests in the file system and make system performance more consistent (details in Section 3.4).

3.2 2-D Data Allocation

The first challenge as described in Section 2.2 is the conflict between data grouping in the file system and internal parallelism in the flash device. Intuitively, the easiest way for file system data groups to be aligned to flash blocks, is using block granularity striping in the FTL. The block striping maintains data grouping while parallelizing data groups to different channels in block units. However, the block-unit striping fails to fully exploit the internal parallelism, since the data within one group needs to be accessed sequentially in the same flash block. We observe in our systems that block-unit striping has significantly lower performance than page-unit striping (i.e., striping in page granularity), especially in the small synchronous write situations, like mail server. Some co-designs employ a super block unit which contains flash blocks from different flash channels [42, 12, 29]. The writes can be striped in a page granularity within the super block and different data groups are maintained in different super blocks. However, the large recycle size of the super block incurs higher overhead in the garbage collection. Therefore, we propose 2-D allocation in ParaFS that uses small allocation units to fully exploit channel-level parallelism while keeping effective data grouping. Tabel 1 summaries the characteristics of these data allocation schemes and compares them with 2-D allocation used in ParaFS.

Figure 4 shows the 2-D allocation in ParaFS. With the device information from S-FTL, ParaFS is able to allocate and recycle space that aligned to the flash memory below. The data space is divided into multiple regions. A region is an abstraction of a flash channel in the device. The number and the size of regions equal to those of flash channels. Each region contains multiple segments and each segments contains multiple data pages. The segment is the unit of allocation and garbage collection

Table 1: Data Allocation Schemes

	Parallelism		Garbage Collecion		
	Stripe Granularity	Parallelism Level	GC Granulariy	Grouping Effect	GC Overhead
Page Stripe	Page	High	Block	No	High
Block Stripe	Block	Low	Block	Yes	Low
Super Block	Page	High	Multiple Blocks	Yes	Medium
2-D Allocation	Page	**High**	Block	Yes	**Low**

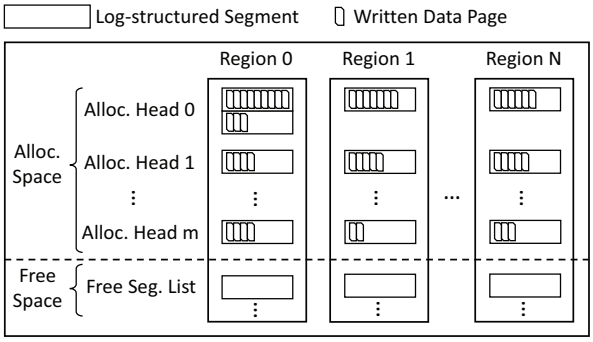


Figure 4: 2-D Allocation in ParaFS

in ParaFS. Its size and address are aligned to the physical flash block. The data page in the segments represents the I/O unit to the flash and is aligned to the flash page. There are multiple allocator heads and a free segment list in each region. The allocator heads point to data groups with different hotness within the region. They are assigned a number of free segments and they allocate free pages to the written data with different hotness, in a log-structured way. The free segments of the region are maintained in the free segment list.

The 2-D allocation consists of channel-level dimension and hotness-level dimension. ParaFS delays space allocation until data persistence. Updated data pages are first buffered in memory. When a write request is going to be dispatched to the device, ParaFS starts the 2-D scheme to allocate free space for the request.

First, in the channel-level dimension, ParaFS divides the write request into pages, and then stripes these pages over different regions. Since the regions match the flash channels one-to-one, the data in the write request is sent to different flash channels in a page-size granularity, so as to exploit the channel-level parallelism in the FS level. After dividing and striping the data of the write request over regions, the allocation process goes to the second dimension.

Second, in the hotness-level dimension, ParaFS groups data pages into groups with different hotness in a region, and sends the divided write requests to their proper groups. There are multiple allocator heads with different hotness in the region. The allocator head, which has similar hotness to the written data, is selected. Finally, the selected allocator head allocates a free page to the written data. The data is sent to the device with the address of the allocated page. Since the segments of allocator heads are aligned to the flash blocks, the data pages with different hotness are assigned to different allocator heads, thereby placed in separated flash blocks. The hotness-level dimension ensures the effective data grouping physically.

In implementation, ParaFS uses six allocators in each region. The data is divided into six kinds of hotness, i.e., hot, warm, and cold, respectively for metadata and regular data. The hotness classification uses the same hints as in F2FS [28]. Each allocator is assigned ten segments each time. The address of the segments and the offset of the allocator heads are recorded in the FS checkpoint in case of system crash.

Crash Recovery. After the crash, legacy log-structured file systems will recover itself to the last checkpoint, which maintains the latest consistent state of the file system. Some file systems[33][28] will do the roll-forward recovery to restore changes after last checkpoint. ParaFS differs slightly during the recovery. Since ParaFS directly accesses and erases flash memory through statically mapped S-FTL, it has to detect the written pages after last checkpoint. These written pages need to be detected so that the file system does not overwrite these pages, which otherwise causes overwrite errors in flash memory.

To solve this problem, ParaFS employs an *update window* similar to that in OFSS [33]. An update window consists of segments from each region that are assigned to allocator heads after last checkpoint. Before a set of segments are assigned to allocator heads, their addresses are recorded first. During recovery, ParaFS first recovers itself to the last checkpoint. Then, ParaFS does the roll-forward recovery with the segments in the update window in each region, since all written pages after last checkpoint fall in this window. After recovering the valid pages in the window, other pages will be considered as invalid, and they will be erased by the GC threads in the future.

3.3 Coordinated Garbage Collection

The second challenge as described in Section 2.2 is the mismatch of the garbage collection processes in FS and FTL levels. Efforts in either side are ineffective in reclaiming free space. ParaFS coordinates garbage collection in the two levels and employs multiple GC threads to leverage the internal parallelism. This tech-

nique improves both the GC efficiency (i.e., the percent of invalid pages in victim blocks) and the space reclaim efficiency (i.e., time used for GC).

The coordinated garbage collection contains the GC process from FS level and FTL level. We use the terms "paraGC" and "flashGC" to distinguish them. In the FS level, the paraGC is triggered when free space drops below a threshold (i.e., foreground paraGC) or file system is idle (i.e., background paraGC). The unit of garbage collection is the segment, as we described in Section 3.2. The foreground paraGC employs greedy algorithm to recycle the segments quickly and minimize the latency of I/O threads. The background paraGC uses cost-benefit algorithm [38] that selects victim segments not only based on the number of invalid data pages, but also their "age". When the paraGC thread is triggered, it first selects victim segments using the algorithms above. Then, the paraGC thread migrates the valid pages in the victim segments to the free space. If the migration is conducted by foreground paraGC, these valid pages are striped to the allocator heads with similar hotness in different regions. If the migration is conducted by background paraGC, these valid pages are considered to be cold, and striped to the allocator heads with low hotness. After the valid pages are written to the device, the victim segments are marked erasable and they will be erased after checkpoint. The background paraGC exits after the migration while the foreground paraGC does the checkpoint and sends the erase requests to S-FTL by *trim*.

In the FTL level, the block recycling is simplified, since the space management and garbage collection are moved from FTL level to FS level. The segments in the FS level match the flash blocks one to one. After paraGC migrates valid pages in the victim segments, the corresponding flash blocks can be erased without additional copies. When S-FTL receives *trim* commands from the ParaFS, flashGC locates the flash blocks that victim segments mapped to, and directly erases them. After the flash blocks are erased, S-FTL informs the ParaFS by the callback function of the requests. The coordinated GC migrates the valid pages in FS level, and invalidates the whole flash block to the S-FTL. No migration overhead is involved during the erase process in the S-FTL.

Coordinated GC also reduces the over-provisioning space of the device and brings more user available capacity. Traditional FTLs need to move the valid pages from the victim blocks before erasing. They keeps large over-provisioning space to reduce the overhead of garbage collection. The spared space in FTL decreases the user visible capacity. Since the valid pages are already moved by ParaFS, the flashGC in S-FTL can directly erase the victim blocks without any migration. S-FTL needn't spare space for garbage collection. The over-provisioning space of S-FTL is much smaller than the traditional ones. The spared blocks are only used for bad block remapping and block mapping table storage. S-FTL also needn't track and maintain the flash page status and flash block utilization, which also reduces the size of spared space.

ParaFS optimizes the foreground GC by employing multiple GC threads. Legacy log-structured file systems perform foreground GC in one thread [28], or none [27]. Since all operations are blocked during checkpointing. Multiple threads cause frequent checkpoint and decrease the performance severely. However, one or less foreground GC thread is not enough under write intensive workloads which consume the free segments quickly. ParaFS assigns one GC thread to each region and employs an additional manager thread. The manager checks the utilization of each region, and wakes up the GC thread of the region when it's necessary. After GC threads migrate the valid data pages, the manager will do the checkpoint, and send the erase requests asynchronously. This optimization avoids multiple checkpoints, and accelerates the segment recycling under heavy writes.

3.4 Parallelism-Aware Scheduling

The third challenge as described in Section 2.2 is the performance spikes. To address this challenge, ParaFS proposes to schedule the read/write/erase requests in the file system level while exploiting the internal parallelism. In ParaFS, the 2-D allocation selects the target flash channels for the write requests, and the coordinated GC manages garbage collection in the FS level. These two techniques offer an opportunity for ParaFS to optimize the scheduling on read, write and erase requests. ParaFS employs parallelism-aware scheduling to provide more consistent performance under heavy write traffic.

Parallelism-aware scheduling consists of request dispatching phase and request scheduling phase. In the dispatching phase, it optimizes the write requests. The scheduler maintains a request queue for each flash channel of the device, shown in Figure 3. The target flash channels of read and erase requests are fixed, while the target channel of writes can be adjusted due to the late allocation. In the channel-level dimension of 2-D allocation, ParaFS splits the write requests into data pages, and selects a target region for each page. The region of ParaFS and the flash channel below are one-to-one correspondence. Instead of a Round-Robin selection, ParaFS selects the region to write, whose corresponding flash channel is the least busy in the scheduler. Due to the asymmetric read and write performance of the flash drive, the scheduler assigns different weights

(W_{read}, W_{write}) to read and write requests in the request queues. The weights are obtained by measuring the corresponding latency of the read and write requests. Since the write latency is 8× of the read latency in our device, the W_{read} and W_{write} are set to 1 and 8 in the evaluation. The channel with the smallest weight calculated by the Formula 1 will be selected. $Size_{read}$ and $Size_{write}$ represent the size of read and write requests in the request queue. The erase requests in the request queue are not considered in the formula, because the time, when they are sent to the device, is uncertain.

$$W_{channel} = \sum(W_{read} \times Size_{read}, W_{write} \times Size_{write}) \quad (1)$$

In the scheduling phase, the scheduler optimizes the erase requests scheduling. Considering the fairness, the parallelism-aware scheduler assigns a time slice for read requests and a same-size slice for write/erase requests. The scheduler works on each request queue individually. In the read slice, the scheduler schedules read requests to the channel. When the slice ends or no read request is left in the queue, the scheduler determines to schedule write or erase request in next slice by Formula 2. The f is the percent of free blocks in the flash channel. N_e is the percent of flash channels that are processing the erase requests at this moment. The a and b represents the weight of these parameters.

$$e = a \times f + b \times N_e \quad (2)$$

If e is higher than 1, the scheduler sends write requests in the time slice. Otherwise, the scheduler sends erase requests in that time slice. After the write/erase slice, the scheduler turns to read slice again. In current implementation, a and b are respectively set to 2 and 1, which implies that erase requests are scheduled only after the free space drops below 50%. This scheme gives higher priority to erase requests when the free space is not enough. Meanwhile, it prevents too many channels erasing at the same time, which helps to ease the performance wave under heavy write traffic.

With the employment of these two optimizations, the parallelism-aware scheduler helps to provide both high and consistent performance.

4 Evaluation

In this section, we evaluate ParaFS to answer the following three questions:

1. How does ParaFS perform compared to other file systems under light write traffic?
2. How does ParaFS perform under heavy write traffic? And, what are the causes behind?
3. What are the benefits respectively from the proposed optimizations in ParaFS?

4.1 Experimental Setup

In the evaluation, we compare ParaFS with Ext4 [2], BtrFS [1] and F2FS [28], which respectively represent the in-place-update, copy-on-write, and flash-optimized log-structured file systems. ParaFS is also compared to a revised F2FS (annotated as F2FS_SB) in addition to conventional F2FS (annotated as F2FS). F2FS organizes data into segment, which is the GC unit in FS and is the same size as flash block. F2FS_SB organizes data into super blocks. A super block consists of multiple adjacent segments. The number of the segments equals to the number of channels in the flash device. In F2FS_SB, a super block is the unit of allocation and garbage collection, and can be accessed in parallel.

We customize a raw flash device to support programmable FTLs. Parameters of the flash device are listed in Table 2. To support the file systems above, we implement a page-level FTL, named PFTL, based on DFTL [17] with lazy indexing technique [33]. PFTL stripes updates over different channels in a page size unit, to fully exploit the internal parallelism. S-FTL is implemented based on PFTL. It removes the allocation function, replaces the page-level mapping with static block-level mapping, and simplifies the GC process, as described in Section 3.1. In both FTLs, the number of flash channels and the capacity of the device are configurable. With the help of these FTLs, we collect the information about flash memory operations, like the number of erase operations and the number of pages migrated during garbage collection. The low-level information is helpful for comprehensive analysis of file system implications.

Table 2: Parameters of the Customized Flash Device

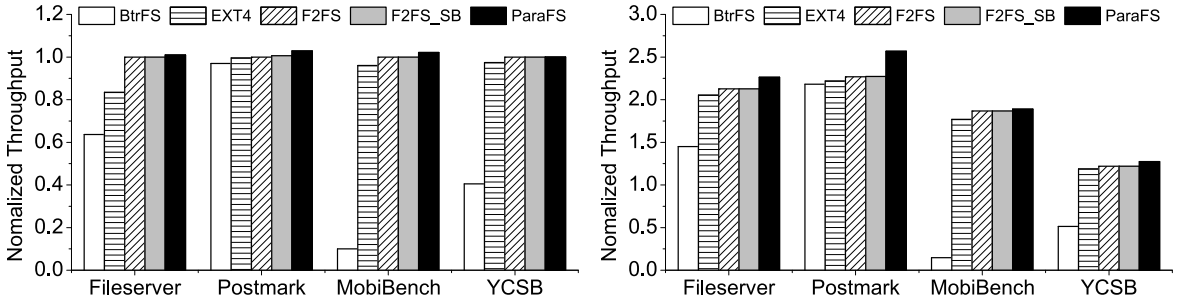
Host Interface	PCIe 2.0 x8
Number of Flash Channel	34
Capacity per Channel	32G
NAND Type	25nm MLC
Page Size	8KB
Block Size	2MB
Read Bandwidth per Channel	49.84 MB/s
Write Bandwidth per Channel	6.55 MB/s

The experiments are conducted on an X86 server with Intel Xeon E5-2620 processor, clocked at 2.10GHz, and 8G memory of 1333MHz. The server runs with Linux kernel 2.6.32, which is required by the customized flash device. For the target system, we back-port F2FS from the 3.15-rc1 main-line kernel to the 2.6.32 kernel. ParaFS is implemented as a kernel module based on F2FS, but differs in data allocation, garbage collection, and I/O scheduling.

Workloads. Table 3 summarizes the four workloads used in the experiments. Two of them run directly on file

Table 3: Workload Characteristics

Name	Description	# of Files	I/O size	Threads	R/W	fsync
Fileserver	File server workload: random read and write files	60,000	1MB	50	33/66	N
Postmark	Mail server workload: create, delete, read and append files	10,000	512B	1	20/80	Y
MobiBench	SQLite workload: random update database records	N/A	4KB	10	1/99	Y
YCSB	MySQL workload: read and update database records	N/A	1KB	50	50/50	Y



(a) channel = 8 (b) channel = 32

Figure 5: Performance Evaluation (Light Write Traffic)

systems and the other two run on databases. Fileserver is a typical pre-defined workload in Filebench [3] to emulate the I/O behaviors in file servers. It creates, deletes and randomly accesses files in multiple threads. Postmark [26] emulates the behavior of mail servers. Transactions, including create, delete, read and append operations, are performed to the file systems. Mobibench [19] is a benchmark tool for measuring the performance of file IO and DB operations. In the evaluation, it issues random update operations to the SQLite[9], which runs with FULL synchronous and WAL journal mode. YCSB [14] is a framework to stress many popular data serving systems. We use the workload-A of YCSB on MySql [7], which consists of 50% random reads and 50% random updates.

4.2 Evaluation with Light Write Traffic

In this section, we compare ParaFS with other file systems under light write traffic. We choose 8 and 32 channels for this evaluation, which respectively represent SATA SSDs [6, 4, 5] and PCIe SSDs [5, 41]. The device capacity is set to 128GB.

Figure 5 shows the throughput of evaluated file systems, and results are normalized against F2FS's performance in 8-channel case. From the figure, we have two observations.

(1) ParaFS outperforms other file systems in all cases, and achieves 13% higher over F2FS for postmark workload in the 32-channel case. In the evaluated file systems, BtrFS performs poorly, especially for database benchmarks that involve frequent syncs. The update propagation (i.e., "wandering tree" problem [38]) of copy-on-write brings intensive data writes in BtrFS during the sync calls (7.2× for mobibench, 3.0× for YCSB). F2FS mitigates this problem by updating the metadata in place [28]. Except BtrFS, other file systems perform roughly similar under light write traffic. F2FS only outperforms Ext4 by 3.5% in fileserver with 32 channels, which is consistent to the F2FS evaluations on PCIe SSD[28]. The performance bottleneck appears to be moved, due to the fast command processing and high random access ability of the PCIe drive. F2FS_SB shows nearly the same performance to F2FS. Since PFTL stripes the requests over all flash channels with a page-size unit, larger allocation unit in F2FS_SB doesn't gain more benefit in parallelism. The GC impact of larger block recycling is also minimized due to the light write pressure. The impact will be seen under heavy write traffic evaluations in Section 4.3. ParaFS uses cooperative designs in both FS and FTL levels, eliminates the duplicate functions, and achieves the highest performance.

(2) Performance gains in ParaFS grow when the number of channels is increased. Comparing the two figures in Figure 5, all file systems have their performance improved when the number of channels is increased. Among them, ParaFS achieves more. It outperforms F2FS averagely by 1.53% in 8-channel cases and 6.29% in 32-channel cases. This also evidences that ParaFS spends more efforts in exploiting the internal parallelism.

In all, ParaFS has comparable or better performance than the other evaluated file systems when the write traffic is light.

4.3 Evaluation with Heavy Write Traffic

Since ParaFS is designed to address the problems of data grouping and garbage collection while exploiting

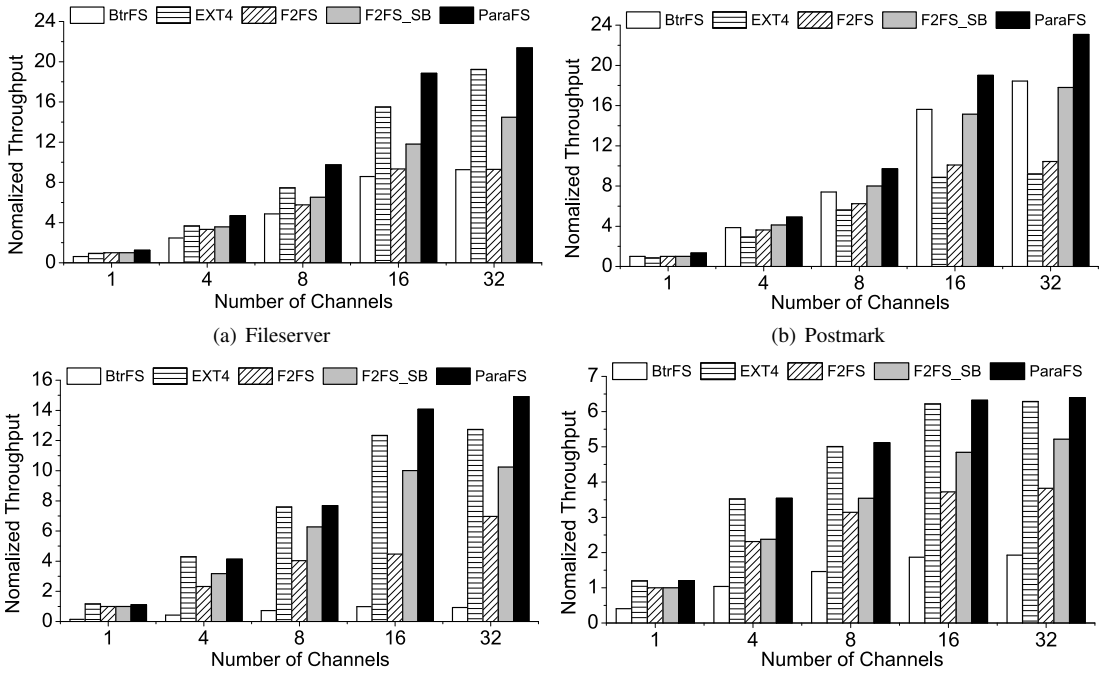


Figure 6: Performance Evaluation (Heavy Write Traffic)

the internal parallelism, evaluations using heavy write traffic are much more important. In this evaluation, we limit the capacity of flash device to 16GB and increase the benchmark sizes. The write traffic sizes in the four evaluated benchmarks are set to $2\times \sim 3\times$ of the flash device capacity, to trigger the garbage collection actions.

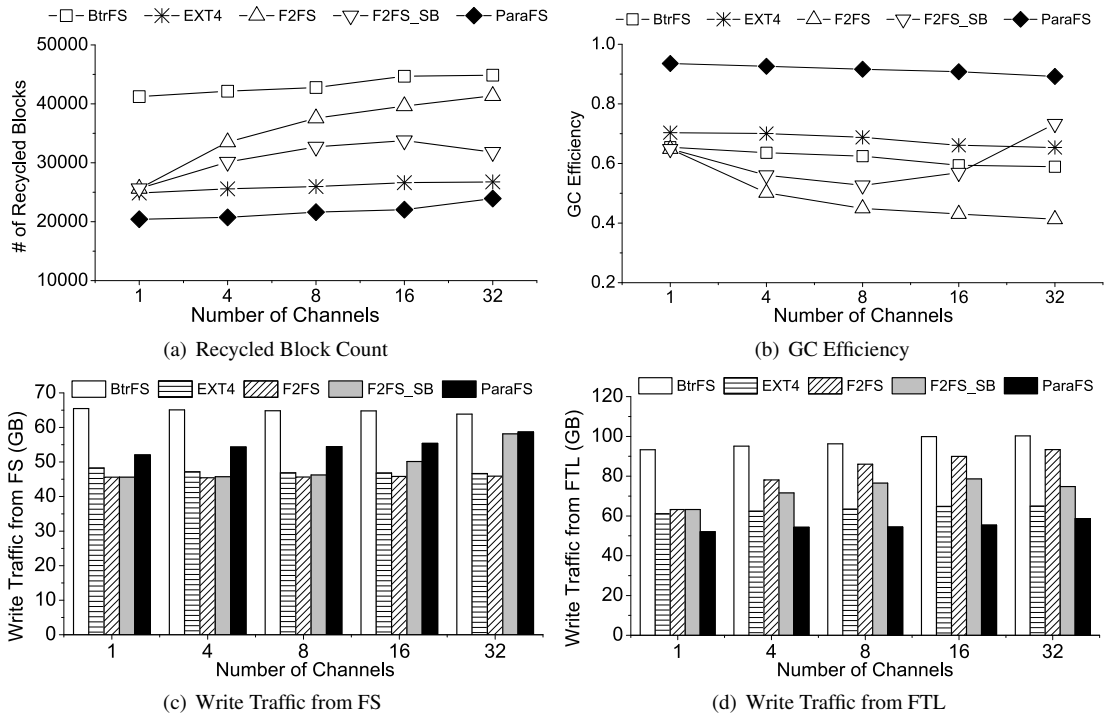
4.3.1 Performance

Figure 6 shows the throughput of evaluated file systems for the heavy write traffic cases, with the number of channels varied from 1 to 32. Results are normalized against F2FS's throughput in 1-channel case. From the figure, we have three observations.

(1) Ext4 outperforms F2FS for three out of the four evaluated workloads, which is different from that in the light write traffic evaluation. The performance gap between Ext4 and F2FS tends to be wider with more flash channels. The reason why the flash-optimized F2FS file system has worse performance than Ext4 is the side effects of internal parallelism. In F2FS, the hot/cold data grouping and the aligned segments are broken when data pages are distributed to multiple channels in the FTL. Also, the invalidation of a page is known in the FTL only after it is recycled in the F2FS, due to the no in-place update. Unfortunately, a lot of invalid pages have been migrated during garbage collection before their statuses are passed to the FTL. Both reasons lead to high GC overhead in FTL, and the problem gets more serious with increased parallelism. In Ext4, the in-place update pattern is more accurate in telling FTLs the page invalidation than F2FS. The exceptional case is the postmark workload, which contains a lot of create and delete operations. Ext4 spreads the inode allocation in all of the block groups for load balance. This causes the invalid pages distributed evenly in the flash blocks and results in higher garbage collection overhead than F2FS (18.5% higher on average). In general, the observation that flash-optimized file system is not good at exploiting internal parallelism under heavy write traffic motivates our ParaFS design.

(2) F2FS_SB shows improved performance than F2FS for the four evaluated workloads, and the improvement grows with more channels. This is also different from results in the light write traffic evaluation. The performance of F2FS improves quickly when the number of channels is increased from 1 to 8, but the improvement is slowed down afterward. For fileserver, postmark and YCSB workloads, F2FS gains little improvement in the 32-channel case over the 16-channel case. The main cause is the increasing GC overhead, which will be seen in next section. In contrast, allocation and recycling in super block units of F2FS_SB ease the GC overhead caused by unaligned segments. The *trim* commands sent by F2FS_SB contain larger address space, and are more effective in telling the invalid pages to the FTL. However, selecting victims in larger units also decreases the GC efficiency. As such, the internal parallelism using super block methods [12, 42] is still not effective.

(3) ParaFS outperforms other file systems in all cases.



(a) Recycled Block Count

(b) GC Efficiency

(c) Write Traffic from FS

(d) Write Traffic from FTL

Figure 7: Garbage Collection and Write Traffic Evaluation (Fileserver)

ParaFS outperforms Ext4 from 1.0× to 1.7× in the 8-channel case, and to 2.5× in the 32-channel case. ParaFS outperforms F2FS from 1.6× to 1.9× in the 8-channel case, and from 1.7× to 3.1× in the 32-channel case. ParaFS outperforms F2FS_SB from 1.2× to 1.5× in both cases. ParaFS keeps the aligned flash block erase units while using page-unit striping in 2-D allocation. The coordinated multi-threaded GC is also helpful in reducing the GC overhead. And thus, ParaFS is effective in exploiting the internal parallelism under heavy write traffic.

4.3.2 Write Traffic and Garbage Collection

To further understand the performance gains in ParaFS, we collect the statistics of garbage collection and write traffic from both FS and FTL levels. We select the fileserver workload as an example due to space limitation. The other workloads have similar patterns and are omitted. For fairness, we revise the fileserver benchmark to write fixed-size data. In the evaluation, the write traffic size in fileserver is set to 48GB, and the device capacity is 16GB.

Figure 7(a) and Figure 7(b) respectively give the recycled block count and the garbage collection efficiency of evaluated file systems with varied number of flash channels. The recycled block count is the number of flash blocks that are erased in flash memory. The GC efficiency is measured using the average percentage of invalid pages in a victim flash block.

ParaFS has the lowest garbage collection overhead and highest GC efficiency among all evaluated file systems under four benchmarks. For the fileserver evaluation, it achieves the lowest recycled block count (62.5% of F2FS on average) and the highest GC efficiency (91.3% on average). As the number of channels increases, the number of recycled blocks in F2FS increases quickly. This is due to the unaligned segments and uncoordinated GC processes of both sides (as analyzed in Section 4.3.1). It is also explained with the GC efficiency degradation in Figure 7(b). The GC efficiency of Ext4, BtrFS, ParaFS trends to drop a little with more flash channels. Because the adjacent pages are more scattered when the device internal parallelism increases, and they tend to be invalidated together. F2FS_SB acts different from other file systems. In F2FS_SB, when the number of channels is increased from 8 to 32, the number of recycled blocks decreases and the GC efficiency increases. The reason is that the super block has better data grouping and alignments, and this advantage becomes increasingly evident with higher degree of parallelism. F2FS_SB also triggers FS-level GC threads more frequently with larger allocation and GC unit. More *trim* commands with larger address space help to decrease the number of invalid pages migrated by GC process in the FTL level. ParaFS further utilizes fine-grained data grouping and GC unit, and has the lowest garbage collection overhead.

Figure 7(c) shows the write traffic that file systems write to FTLs. Figure 7(d) shows the write traffic

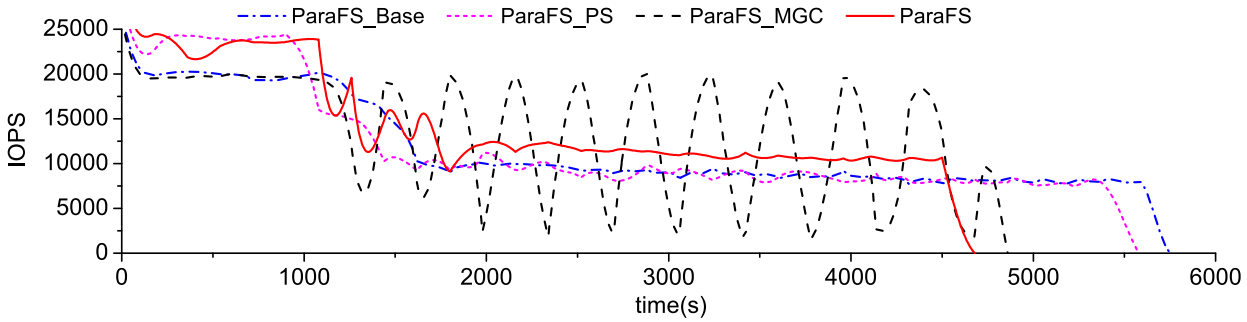


Figure 8: Performance Consistency Evaluation

that FTLs write to the flash memory. Write traffic in either level comes from not only the file system data and metadata but also the page migrated during garbage collection in FS or FTL level. The write traffic from FS in Ext4, BtrFS and F2FS are stable for different parallelism levels, because the increase of the flash channels in the device is transparent to them. ParaFS writes more in the file system than Ext4, F2FS and F2FS_SB, but less than them in the FTL. Because all the page migration during the garbage collection is done in the FS level. Similarly, F2FS_SB has higher write traffic from FS and lower from FTL than F2FS, as the number of channels is increased from 8 to 32. This results from the improved GC efficiency as mentioned above. The FTL write traffic in F2FS is higher than F2FS_SB, which explains why F2FS_SB has better performance than F2FS in Figure 6. ParaFS coordinates garbage collections in the two levels, and is more effective in space recycling. Compared to F2FS, ParaFS decreases the write traffic to the flash memory by 31.7% $\sim$ 54.7% in 8-channel case, and 37.1% $\sim$ 58.1% in 32-channel case. Compared to F2FS_SB, ParaFS decreases it by 14.9% $\sim$ 48.4% in 8-channel case, and 15.7% $\sim$ 32.5% in 32-channel case.

4.4 Performance Consistency Evaluation

To evaluate the performance consistency in ParaFS, we monitor the throughput wave during each run of experiments. ParaFS aims at more consistent performance using multi-threaded GC and parallelism-aware scheduling. In this evaluation, we use four versions of ParaFS. The baseline (annotated as ParaFS_Base) is the ParaFS version without the above-mentioned two optimizations. ParaFS_PS and ParaFS_MGC respectively stand for the version with parallelism-aware scheduling and multi-threaded GC. ParaFS is the fully-functioned version.

The fileserver and postmark have different phases in the evaluation, which also cause fluctuation in the aggregate throughput (in terms of IOPS). We choose mobibench as the candidate for performance consistency evaluation. Mobibench performs random asynchronous reads and writes to pre-allocated files. In this evaluation, the write traffic of mobibench is set to be 2$\times$ of the device capacity.

Figure 8 shows the process of each run using four versions of ParaFS. We compare ParaFS_MGC and ParaFS_Base to analyse the impact of multi-threaded GC. ParaFS_MGC and ParaFS_Base have similar performance in the first 1000 seconds, during which no garbage collection is involved. After that, the performance of ParaFS_MGC waves. The performance peaks appear after the GC starts in multiple threads. ParaFS_MGC finishes the experiments earlier than ParaFS_Base by 18.5%.

The parallelism-aware scheduling contains two major methods, the write request dispatching and the erase request scheduling. The effectiveness of write request dispatching can be seen by comparing ParaFS_PS and ParaFS_Base. For the first 1000 seconds when there is no garbage collection, ParaFS_PS outperforms ParaFS_Base by nearly 20%. This benefit comes from the write dispatching in the parallelism-aware scheduling technique, which allocates pages and sends requests to the least busy channels. The effectiveness of erase request scheduling can be observed between ParaFS and ParaFS_MGC. In the latter part of each run when the GC processes are frequently triggered, ParaFS using parallelism-aware scheduling performs more consistently than ParaFS_MGC.

In conclusion, the FS-level multi-threaded garbage collection as implemented in ParaFS is more effective in reclaiming free space, and the FS-level parallelism-aware scheduling makes performance more consistent.

5 Related Work

Data Allocation. In the FTL or flash controller, internal parallelism has been extensively studied. Recent researches [13, 18, 21] have conducted extensive experiments on page allocation with different levels (i.e., channel-level, chip-level, die-level and plane-level) of internal parallelism. Gordon [12] introduces a 2-D striping to leverage both channel-level and die-level parallelism.

But note that, 2-D striping in Gordon is different from 2-D data allocation in ParaFS. 2-D striping is designed in the flash controller, which places data to leverage multiple levels of parallelism. 2-D data allocation is designed in file system, which organizes data into different groups using metrics of parallelism and hotness. In addition, aggressive parallelism in the device level scatters data addresses, breaking up the data organization in the system level. P-OFTL [42] has pointed out this problem and found that increased parallelism leads to higher garbage overhead, which in turn can decrease the overall performance.

In the file system level, DirectFS [20] and Nameless Write [45] propose to remove data allocation functions from file systems and leverage the data allocations in the FTL or storage device, which can have better decisions with detailed knowledge of hardware internals. OFSS [33] proposes an object-based FTL, which enables hardware/software codesign with both knowledge of file semantics and hardware internals. However, these file systems pay little attention to internal parallelism, which is the focus of ParaFS in this paper.

Garbage Collection. Garbage collection has a strong impact on system performance for log-structured designs. Researchers are trying to pass more file semantics to FTLs to improve GC efficiency. For instance, *trim* is a useful interface to inform FTLs the data invalidation, in order to reduce GC overhead in migrating invalid pages. Also, Kang et al. [25] found that FTLs can have more efficient hot/cold data grouping, which further reduces GC overhead, if the expected lifetime of written data is passed from file systems to FTLs. In addition, Yang et al. [44] found log-structured designs in both levels of system software and FTLs have semantic gaps, which make garbage collection in both levels inefficient. In contrast, ParaFS proposes to bridge the semantic gap and coordinate garbage collection in the two levels.

In the file system level, SFS [34] uses a sophisticated hot/cold data grouping algorithm using both access count and age of the block. F2FS [28] uses a static data grouping method according to the file and data types. However, these grouping algorithms suffer when grouped data are spread out in the FTL. Our proposed ParaFS aims to solve this problem and keep physical hot/cold grouping while exploiting the internal parallelism.

A series of research works use large write block size to align the flash block and decrease the GC overhead[40, 31, 30, 35]. RIPQ[40] and Pannier[31] aggregate small random writes in memory, divide them into groups according to the hotness, and evict the groups in flash block size. Nitro[30] deduplicates and compresses the writes in RAM and evicts them in flash block size. Nitro proposes to modify the FTL to support block-unit striping that ensures the effective of the block-size write optimization. SDF[35] employs block-unit striping which is tightly coupled with key-value workloads. ParaFS uses page-size I/O unit and aims at file system workloads.

I/O Scheduling. In flash storage, new I/O scheduling policies have been proposed to improve utilization of internal parallelism [24, 23, 16] or fairness [37, 39]. These scheduling policies are designed in the controller, the FTL or the block layer. In these levels, addresses of requests are determined. In comparison, system-level scheduling can schedule write requests before data allocation, which is flexible.

LOCS [41] is a key-value store that schedules I/O requests in the system level upon open-channel SSDs. With the use of log-structured merge tree (LSM-tree), data is organized into data blocks aligned to flash blocks. LOCS schedules the read, write and erase operations to minimize the response time.

Our proposed ParaFS is a file system that schedules I/O requests in the system level. Different from key-value stores, file systems have irregular reads and writes. ParaFS exploits the channel-level parallelism with page-unit striping. Moreover, its goal is in making performance more consistent.

6 Conclusion

ParaFS is effective in exploiting the internal parallelism of flash storage, while keeping physical hot/cold data grouping and low garbage collection overhead. It also takes the parallelism opportunity to schedule read, write and erase requests to make system performance more consistent. ParaFS's design relies on flash devices with customized FTL that exposes physical layout, which can be represented by three values. The proposed design bridges the semantic gap between file systems and FTLs, by simplifying FTL and coordinating functions of the two levels. We implement ParaFS on a customized flash device. Evaluations show that ParaFS outperforms the flash-optimized F2FS by up to 3.1×, and has more consistent performance.

Acknowledgments

We thank our shepherd Haryadi Gunawi and anonymous reviewers for their feedbacks and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61433008, 61502266), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and the China Postdoctoral Science Foundation (Grant No. 2015M580098).

References

[1] Btrfs. http://btrfs.wiki.kernel.org.

[2] Ext4. https://ext4.wiki.kernel.org/.

[3] Filebench benchmark. http://sourceforge.net/apps/mediawiki/filebench.

[4] Intel dc s3500 480gb enterprise ssd review. http://www.tweaktown.com/reviews/5534/intel-dc-s3500-480gb-enterprise-ssd-review/index.html.

[5] Intel ssd 750 pcie ssd review. http://www.anandtech.com/show/9090/intel-ssd-750-pcie-ssd-review-nvme-for-the-client.

[6] Intel x25-m and x18-m mainstream sata solid-state drives. ftp://download.intel.com/newsroom/kits/ssd/pdfs/X25-M_34nm_ProductBrief.pdf.

[7] Mysql. https://www.mysql.com/.

[8] FusionIO Virtual Storage Layer. http://www.fusionio.com/products/vsl, 2013.

[9] SQLite. http://www.sqlite.org/, 2014.

[10] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC)*, Berkeley, CA, 2008. USENIX.

[11] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, page 22. ACM, 2013.

[12] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 217–228, New York, NY, USA, 2009. ACM.

[13] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277. IEEE, 2011.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[15] C. Dirik and B. Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. ACM, 2009.

[16] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H. Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–11. IEEE, 2014.

[17] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 229–240, New York, NY, USA, 2009. ACM.

[18] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 96–107. ACM, 2011.

[19] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won. Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace. *Future Internet*, 5(4):591–610, 2013.

[20] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2010. USENIX.

[21] M. Jung and M. Kandemir. An evaluation of different page allocation strategies on high-speed ssds. In *Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems*, pages 9–9. USENIX Association, 2012.

[22] M. Jung and M. Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of the fifteenth international joint conference on Measurement and modeling of computer systems (SIGMETRICS)*, pages 203–216. ACM, 2013.

[23] M. Jung and M. T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 524–535. IEEE, 2014.

[24] M. Jung, E. H. Wilson III, and M. Kandemir. Physically addressed queueing (paq): improving parallelism in solid state disks. In *Proceedings of the 39th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 404–415, 2012.

[25] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*, pages 13–13. USENIX Association, 2014.

[26] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997.

[27] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[28] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, Feb. 2015. USENIX.

[29] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST)*, pages 339–353, Santa Clara, CA, 2016. USENIX Association.

[30] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of 2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 501–512, Philadelphia, PA, June 2014. USENIX Association.

[31] C. Li, P. Shilane, F. Douglis, and G. Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference*, pages 50–62, Vancouver, Canada, 2015. ACM.

[32] Y. Lu, J. Shu, and W. Wang. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, pages 75–88, Berkeley, CA, 2014. USENIX.

[33] Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2013. USENIX.

[34] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

[35] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 471–484, New York, NY, USA, 2014. ACM.

[36] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 301–311. IEEE, 2011.

[37] S. Park and K. Shen. Fios: a fair, efficient flash i/o scheduler. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

[38] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[39] K. Shen and S. Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC)*, pages 67–78, Berkeley, CA, 2013. USENIX.

[40] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 373–386, Santa Clara, CA, Feb. 2015. USENIX Association.

[41] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, pages 16:1–16:14, New York, NY, USA, 2014. ACM.

[42] W. Wang, Y. Lu, and J. Shu. p-OFTL: an object-based semantic-aware parallel flash translation layer. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, page 157. European Design and Automation Association, 2014.

[43] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

[44] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Dont stack your log on my log. In *USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[45] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, Berkeley, CA, 2012. USENIX.

FastCDC: a Fast and Efficient Content-Defined Chunking Approach for Data Deduplication

Wen Xia[†,‡], Yukun Zhou[†], Hong Jiang[§], Dan Feng[¶,†,*], Yu Hua[¶,†], Yuchong Hu[†], Yucheng Zhang[†], Qing Liu[†]

[†]School of Computer, Huazhong University of Science and Technology [‡]Sangfor Technologies Co., Ltd.

[§]University of Texas at Arlington [¶]WNLO, Huazhong University of Science and Technology

Abstract

Content-Defined Chunking (CDC) has been playing a key role in data deduplication systems in the past 15 years or so due to its high redundancy detection ability. However, existing CDC-based approaches introduce heavy CPU overhead because they declare the chunk cut-points by computing and judging the rolling hashes of the data stream byte by byte. In this paper, we propose FastCDC, a Fast and efficient CDC approach, that builds and improves on the latest Gear-based CDC approach, one of the fastest CDC methods to our knowledge. The key idea behind FastCDC is the combined use of three key techniques, namely, simplifying and enhancing the hash judgment to address our observed challenges facing Gear-based CDC, skipping sub-minimum chunk cut-point to further speed up CDC, and normalizing the chunk-size distribution in a small specified region to address the problem of the decreased deduplication ratio stemming from the cut-point skipping. Our evaluation results show that, by using a combination of the three techniques, FastCDC is about 10× faster than the best of open-source Rabin-based CDC, and about 3× faster than the state-of-the-art Gear- and AE-based CDC, while achieving nearly the same deduplication ratio as the classic Rabin-based approach.

1 Introduction

Data deduplication, an efficient approach to data reduction, has gained increasing attention and popularity in large-scale storage systems due to the explosive growth of digital data. It eliminates redundant data at the file- or chunk-level and identifies duplicate contents by their cryptographically secure hash signatures (e.g., SHA1 fingerprint). According to deduplication studies conducted by Microsoft [12, 23] and EMC [30, 33], about 50% and 85% of the data in their production primary and secondary storage systems, respectively, are redundant and could be removed by the deduplication technology.

In general, chunk-level deduplication is more popular than file-level deduplication because it identifies and removes redundancy at a finer granularity. For chunk-level deduplication, the simplest chunking approach is to cut the file or data stream into equal, fixed-size chunks, referred to as Fixed-Size Chunking (FSC) [27]. Content-Defined Chunking (CDC) based approaches are proposed to address the *boundary-shift* problem facing the FSC approach [25]. Specifically, CDC declares chunk boundaries based on the byte contents of the data stream instead of on the byte offset, as in FSC, and thus helps detect more redundancy for deduplication. According to some recent studies [12, 22, 23, 26], CDC-based deduplication approaches are able to detect about 10-20% more redundancy than the FSC approach.

Currently, the most popular CDC approaches determine chunk boundaries based on the Rabin fingerprints of the content, which we refer to as Rabin-based CDC [8, 11, 25, 28]. Rabin-based CDC is highly effective in duplicate detection but time-consuming, because it computes and judges (against a condition value) the Rabin fingerprints of the data stream byte by byte [11]. A recent study, called QuickSync [9], suggests that CDC is computationally expensive for deduplication based synchronization in mobile cloud storage. In order to speed up the CDC process, other hash algorithms have been proposed to replace the Rabin algorithm for CDC, such as SampeByte [1], Gear [38], and AE [40]. Meanwhile, the abundance of computation resources afforded by multicore and manycore processors [20, 37] or GPU processors [2, 5, 15] has been leveraged for acceleration.

Generally, CDC consists of two distinctive and sequential stages: *(1) hashing* in which fingerprints of the data contents are generated and *(2) hash judgment* in which fingerprints are compared against a given value to identify and declare chunk cut-points. Our previous study of delta compression, Ddelta [38], suggests that the Gear hash (i.e., $fp=(fp\ll 1)+G(b)$, see Section 3) is very efficient as a rolling hash for CDC. To the best of our knowledge, Gear appears to be one of the fastest rolling hash algorithms for CDC at present. However, according to our first observation from empirical and analytical studies, the Gear-based CDC has the potential problem of low *deduplication ratio* (i.e., the percentage of re-

*Corresponding author: dfeng@hust.edu.cn.

dundant data reduced) stemming from its *hash judgment* stage where the sliding window size is very small. Meanwhile, our second observation indicates that the *hash judgment* stage becomes the new performance bottleneck during CDC after the fast Gear [38] is used in the *hashing* stage, because the accelerated hashing stage by Gear, has shifted the bottleneck to the hash judgment stage. Motivated by these two observations and the need to further accelerate the CDC process, we use an approach of enhancing and simplifying the hash judgment to further reduce the CPU operations during CDC.

Our third observation suggests that the predefined minimum chunk size used to avoid generating the very small-sized chunks (e.g., LBFS [25] employs the minimum chunk size of 2KB for Rabin-based CDC) can be employed for *cut-point skipping* during CDC, i.e., judiciously skipping some identified cut-points to eliminate the CDC operations in this region. Enlarging this minimum chunk size can further speed up the chunking process but at the cost of decreasing the deduplication ratio. *This is because many chunks with skipped cut-points are not divided truly according to the data contents (i.e., content-defined).* Thus, we propose a novel normalized Content-Defined Chunking scheme, called **normalized chunking**, that normalizes the chunk-size distribution to a specified region that *is guaranteed to be larger than the minimum chunk size* to effectively address the problem facing the cut-point skipping approach.

Therefore, motivated by the above observations, we proposed FastCDC, a Fast and efficient CDC approach that combines the following three key techniques.

- **Simplified but enhanced hash judgment:** By padding several zero bits into the mask value for the hash-judging statement of the CDC algorithm to enlarge the sliding window size while using the fast Gear hash, FastCDC is able to achieve nearly the same deduplication ratio as the Rabin-based CDC; By further simplifying and optimizing the hash-judging statement, FastCDC minimizes the CPU overhead for the hash judgment stage in CDC.
- **Sub-minimum chunk cut-point skipping:** Our large scale study suggests that skipping the predefined minimum chunk size (used for avoiding small-sized chunks) increases the chunking speed but decreases the deduplication ratio (about 15% decline in the worst case). This motivates us to further enlarge the minimum chunk size to maximize chunking speed while developing a counter measure for the decreased deduplication ratio in the following normalized chunking approach.
- **Normalized chunking:** By selectively changing the number of mask bits '1' in the hash-judging statement of CDC, FastCDC normalizes the chunk-size distribution to a small specified region (e.g., 8KB∼16KB), i.e., the vast majority of the generated chunks fall into this size range, and thus minimizes the number of chunks of either too small or large in size. The benefits are twofold. First, it increases the deduplication ratio by reducing the number of large-sized chunks. Second, it reduces the number of small-sized chunks, which makes it possible to combine with the cut-point skipping technique above to maximize the CDC speed while without sacrificing the deduplication ratio.

Our evaluation results from a large-scale empirical study of CDC, based on seven datasets, demonstrate that FastCDC is about 10× faster than the Rabin-based CDC, and 3× faster than the state-of-the-art Gear- and AE-based CDC, while ensuring a high deduplication ratio.

The rest of the paper is organized as follows. Section 2 presents the necessary background for this research. Section 3 discusses our key observations that motivate the design of FastCDC. Section 4 describes the three key approaches used in FastCDC. Section 5 presents and discusses our experimental evaluation of FastCDC. Section 6 draws conclusions and outlines our future work.

2 Background

Chunking is the first critical step in the operational path of data deduplication, in which a file or data stream is divided into small chunks so that each can be duplicate-identified. Fixed-Size Chunking (FSC) [27] is simple and fast but may face the problem of low deduplication ratio that stems from the *boundary-shift* problem [25, 40]. For example, if one or several bytes are inserted at the beginning of a file, all current chunk cut-points (i.e., boundaries) declared by FSC will be shifted and no duplicate chunks will be detected.

Content-Defined Chunking (CDC) is proposed to solve the *boundary-shift* problem. CDC uses a sliding-window technique on the content of files and computes a hash value (e.g., Rabin fingerprint [25, 28]) of the window. A chunk cut-point is declared if the hash value satisfies some pre-defined condition. As shown in Figure 1, to chunk a file V_2 that is modified from the file V_1, the CDC algorithm can still identify the correct boundary of chunks C_1, C_3, and C_4, whose contents have not been modified. As a result, CDC outperforms FSC in terms of deduplication ratio and has been widely used in backup [33, 42] and primary [12, 23] storage systems.

Although the widely used Rabin-based CDC helps obtain a high deduplication ratio, it incurs heavy CPU overhead [2, 5, 9, 37]. Specifically, in Rabin-based CDC, the Rabin hash for a sliding window containing the byte sequence B_1,B_2,...,B_α is defined as a polynomial A(p):

$$Rabin(B_1, B_2, ..., B_\alpha) = A(p) = \{\sum_{x=1}^{\alpha} B_x p^{\alpha - x}\} mod\ D \quad (1)$$

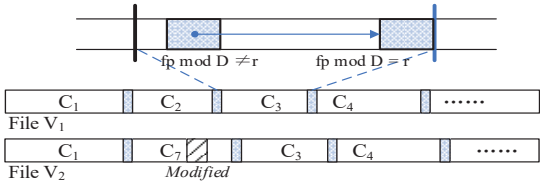


Figure 1: The sliding window technique for the CDC algorithm. The hash value of the sliding window, *fp*, is computed via the Rabin algorithm (this is the *hashing stage* of CDC). If the lowest $log_2 D$ bits of the hash value matches a threshold value r, i.e., $fp \bmod D = r$, this offset (i.e., the current position) is marked as a chunk cut-point (this is the *hash-judging stage* of CDC).

where D is the average chunk size and α is the number of bytes in the sliding window. Rabin hash is a *rolling hash* algorithm since it is able to compute the hash in an iterative fashion, i.e., the current hash can be incrementally computed from the previous value as follows:

$$Rabin(B_2, B_3, ..., B_{\alpha+1}) = \{[Rabin(B_1, ..., B_\alpha) - B_1 P^{\alpha-1}]p + B_{\alpha+1}\} mod S \quad (2)$$

However, Rabin-based CDC is time-consuming because it computes and judges the hashes of the data stream byte by byte, which renders the chunking process a performance bottleneck in deduplication systems. There are many approaches to accelerating the CDC process for deduplication systems and they can be broadly classified as either algorithmic oriented or hardware oriented. We summarize below some of these approaches that represent the state of the art.

Algorithmic-oriented CDC Optimizations. Since the frequent computations of Rabin fingerprints for CDC are time-consuming, many alternatives to Rabin have been proposed to accelerate the CDC process [1, 38, 40]. SampleByte [1] is designed for providing fast chunking for fine-grained network redundancy elimination, usually eliminating duplicate chunks as small as 32-64 bytes. It uses one byte to declare a fingerprint for chunking, in contrast to Rabin that uses a sliding window, and skips $\frac{1}{2}$ of the expected chunk size before chunking to avoid generating extremely small-sized strings or chunks (they called "avoid oversampling"). Gear [38] uses fewer operations to generate rolling hashes by means of a small random integer table to map the values of the byte contents, so as to achieve higher chunking throughput. AE [40] is a non-rolling-hash-based chunking algorithm that employs an asymmetric sliding window to identify extremums of data stream as cut-points, which reduces the computational overhead for CDC. Yu et al. [39] adjust the function for selecting chunk boundaries such that if weak conditions are not met, the sliding window can jump forward, avoiding unnecessary calculation steps.

Hardware-oriented CDC Optimizations. StoreGPU [2, 15] and Shredder [5] make full use of GPGPU's computational power to accelerate popular compute-intensive primitives (i.e., chunking and fingerprinting) in data deduplication. P-Dedupe [37] pipelines deduplication tasks and then further parallelizes the sub-tasks of chunking and fingerprinting with multiple threads and thus achieves higher throughput.

It is noteworthy that there are other chunking approaches trying to achieve a higher deduplication ratio but introduce more computation overhead on top of the conventional CDC approach. TTTD [13] and Regression chunking [12] introduces one or more additional thresholds for chunking judgment, which leads to a higher probability of finding chunk boundaries and decreases the chunk size variance. MAXP [3, 7, 32] treats the extreme values in a fixed-size region as cut-points, which also results in smaller chunk size variance. In addition, Bimodal chunking [17], Subchunk [29], and FBC [21] re-chunk the non-duplicate chunks into smaller ones to detect more redundancy.

For completeness and self-containment we briefly discuss *other relevant deduplication issues* here. A typical data deduplication system follows the workflow of chunking, fingerprinting, indexing, and storage management [14, 19, 34, 42]. The fingerprinting process computes the cryptographically secure hash signatures (e.g., SHA1) of data chunks, which is also a compute-intensive task but can be accelerated by certain pipelining or parallelizing techniques [16, 36, 37]. Indexing refers the process of identifying the identical fingerprints for checking duplicate chunks in large-scale storage systems, which has been well explored in many previous studies [10, 14, 35, 42]. Storage management refers to the storage and possible post-deduplication processing of the non-duplicate chunks and their metadata, including such processes as related to further compression [38], defragmentation [18], reliability [4], security [41], etc. In this paper, we focus on designing a very fast and efficient chunking approach for data deduplication.

3 Observation and Motivation

In this section, we elaborate on and analyze the most relevant state-of-the-art CDC approaches to gain useful insights and observations. Table 1 shows a comparison among the three rolling hash algorithms for CDC, namely, Rabin, Adler, and Gear, which suggests Gear uses far fewer calculation operations than Rabin and Adler, thus being a good rolling hash candidate for CDC.

A good hash function must have a uniform distribution of hash values regardless of the hashed content. As shown in Figure 2, Gear-based CDC achieves this in two key ways: (1) It employs an array of 256 random 64-bit integers to map the values of the byte contents in the sliding window (i.e., the calculated bytes, whose size is the

Name	Pseudocode	Speed
Rabin	$fp = ((fp \hat{} U(a)) << 8) \vert b \hat{} T[fp >> N]$	Slow
Adler	$S_1 += A(b); S_2 += S_1; fp = (S_2 << 16) \vert S_1$	Slow
Gear	$fp = (fp << 1) + G(b)$	Fast

Table 1: The hashing stage of the Rabin-, Adler-, and Gear-based CDC. Here 'a' and 'b' denote contents of the first and last byte of the sliding window respectively, 'N' is the length of the content-defined sliding window, and 'U', 'T', 'A', 'G' denote the predefined arrays [11, 25, 38]. '*fp*' represents the fingerprint of the sliding window.

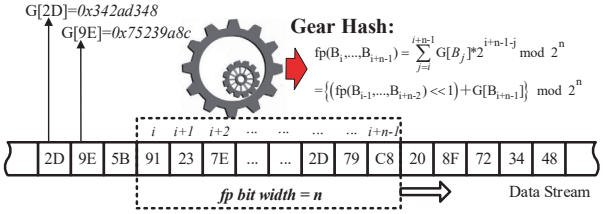


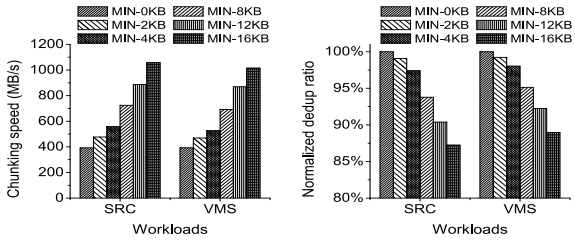
Figure 2: A schematic diagram of the Gear hash.

bit-width of the *fp*); and (2) The addition ("+") operation adds the new byte in the sliding window into Gear hashes while the left-shift ("<<") operation helps strip away the last byte of the last sliding window (e.g., B_{i-1} in Figure 2). This is because, after the "<<" and modulo operations, the last byte B_{i-1} will be calculated into the *fp* as the $(G[B_{i-1}] << n)\ mod\ 2^n$, which will be equal to zero. As a result, Gear generates uniformly distributed hash values by using only three operations (i.e., "+", "<<", and an array lookup), enabling it to move quickly through the data content for the purpose of CDC. Note that the modulo operation is used in the hashing-judging stage as discussed later.

Gear-based CDC is first employed by Ddelta [38] for delta compression, which helps provide a higher delta encoding speed. However, according to our experimental analysis, there are still challenges facing the Gear-based CDC. We elaborate on these issues as follows.

Limited sliding window size. The traditional hash judgment for the Rabin-based CDC, as shown in Figure 1 (i.e., "*fp mod D==r*"), is also used by the Gear-based CDC [38]. But this results in a smaller sized sliding window used by Gear-based CDC since it uses Gear hash for chunking. For example, as shown in Figure 5, the sliding window size of the Gear-based CDC will be equal to the number of the bits used by the mask value. Therefore, when using a mask value of 2^{13} for the expected chunk size of 8KB, the sliding window for the Gear-based CDC would be 13 bytes while that of the Rabin-based CDC would be 48 bytes [25]. The smaller sliding window size of the Gear-based CDC can lead to more chunking position collisions (i.e., randomly marking the different positions as the chunk cut-points), resulting in the decrease in deduplication ratio (see Section 5.2).

The time-consuming hash judgment. Our implementation and in-depth analysis of the Gear-based CDC suggest that *its hash-judging stage accounts for more than 60% of its CPU overhead during CDC after the fast Gear hash is used for fingerprinting*. Thus, there is a lot of room for the optimization of the hash judging stage to further accelerate the CDC process.



(a) Chunking speed (b) Deduplication ratio

Figure 3: Rabin-based CDC performance as a function of the minimum chunk size used for cut-points skipping before chunking. Here we use the average chunk size of 8KB, Intel i7-4770 processor, and the best open-source Rabin algorithm we have access to for the speed test.

Speed up chunking by skipping. Another observation is that the minimum chunk size used for avoiding extremely small-sized chunks, can be also employed to speed up CDC by the cut-point skipping, i.e., eliminating the chunking computation in the skipped region. Figure 3 shows our experimental observation of Rabin-based CDC with two typical workloads of deduplication whose workload characteristics are detailed in Table 2 in Section 5.1. Figure 3 (a) indicates that setting the minimum chunk size for cut-point skipping at $\frac{1}{4}\times\sim 2\times$ of the expected chunk size can effectively accelerate the CDC process. But this approach decreases the deduplication ratio by about 2∼15% (see Figure 3 (b)) since many chunks are not divided truly according to the data contents, i.e., not really content-defined.

The observation suggested in Figure 3 motivates us to consider a new CDC approach that (1) keeps all the chunk cut-points that generate chunks larger than a predefined minimum chunk size and (2) enables the chunk-size distribution to be normalized to a relatively small specified region, an approach we refer to as **normalized chunking** in this paper, as described in Section 4.4.

In summary, the analysis and observation of the Gear-based CDC motivate us to propose FastCDC, a faster CDC approach with a higher deduplication ratio than the Gear-based CDC. The implementation of FastCDC will be detailed in the next section and its effectiveness and efficiency will be demonstrated in Section 5.

4 FastCDC Design and Implementation

4.1 FastCDC Overview

FastCDC is implemented on top of the Gear-based CDC, and aims to provide high performance CDC. Generally,

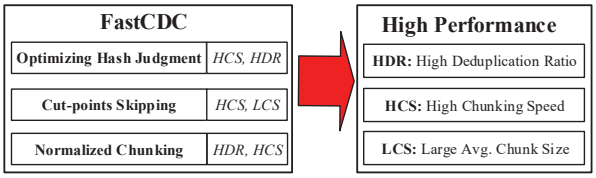


Figure 4: The three key techniques used in FastCDC and their corresponding benefits for high performance CDC.

there are three metrics for evaluating CDC performance, namely, deduplication ratio, chunking speed, and the average generated chunk size. Note that the average generated chunk size may be nearly equal to or larger than the predefined expected chunk size (e.g., 8KB) due to factors such as the detailed CDC methods and datasets. This is also an important CDC performance metric because it reflects the metadata overhead for deduplication indexing, i.e., the larger the generated chunk size is, the fewer the number of chunks and thus the less metadata will be processed by data deduplication. However, it is difficult, if not impossible, to improve these three performance metrics simultaneously because they can be conflicting goals. For example, a smaller average generated chunk size leads to a higher deduplication ratio, but at the cost of lower chunking speed and high metadata overheads. Thus, FastCDC is designed to strike a sensible tradeoff among these three metrics so as to strive for high performance CDC, by using a combination of the three techniques with their complementary features as shown in Figure 4.

- Optimizing hash judgment: using a zero-padding scheme and a simplified hash-judging statement to speed up CDC without compromising the deduplication ratio, as detailed in Section 4.2.
- Sub-minimum chunk cut-point skipping: enlarging the predefined minimum chunk size and skipping cut-points for chunks smaller than that to provide a higher chunking speed and a larger average generated chunk size, as detailed in Section 4.3.
- Normalized chunking: selectively changing the number of mask '1' bits for the hash judgment to approximately normalize the chunk-size distribution to a small specified region that is just larger than the predefined minimum chunk size, ensuring both a higher deduplication ratio and higher chunking speed, as detailed in Section 4.4.

In general, the key idea behind FastCDC is the combined use of the above three key techniques on top of Gear-based CDC, especially employing normalized chunking to address the problem of decreased deduplication ratio facing the cut-point skipping, and thus achieve high performance CDC on the three key metrics.

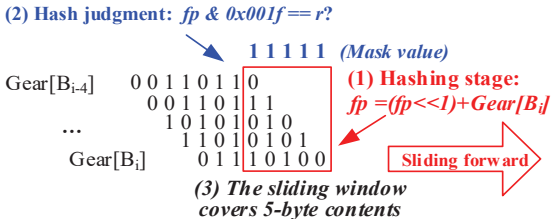


Figure 5: An example of the sliding window technique used in the Gear-based CDC. Here CDC consists of two stages: hashing and hash judgment. The size of the sliding window used for hash judgment is only 5 bytes because of the computation principles of the Gear hash.

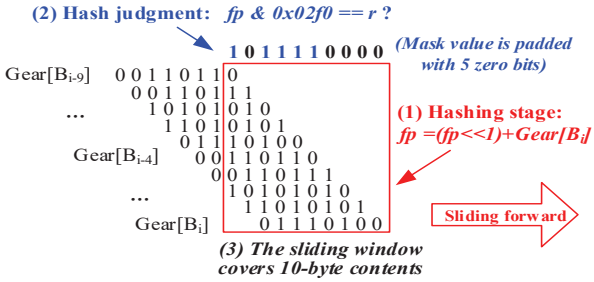


Figure 6: An example of the sliding window technique proposed for FastCDC. By padding y zero bits into the mask value for hash judgment, the size of the sliding window used in FastCDC is enlarged to about $5+y$ bytes, where $y=5$ in this example.

4.2 Optimizing Hash Judgment

In this subsection, we propose an enhanced but simplified hash-judging statement to accelerate the hash judgment stage of FastCDC to further accelerate the chunking process on top of the Gear-based CDC and increase the deduplication ratio to reach that of the Rabin-based CDC. More specifically, FastCDC incorporates two main optimizations as elaborated below.

Enlarging the sliding window size by zero padding. As discussed in Section 3, the Gear-based CDC employs the same conventional hash judgment used in the Rabin-based CDC, where a certain number of the lowest bits of the fingerprint are used to declare the chunk cut-point, leading to a shortened sliding window for the Gear-based CDC (see Figure 5) because of the unique feature of the Gear hash. To address this problem, FastCDC enlarges the sliding window size by padding a number of zero bits into the mask value. As illustrated by the example of Figure 6, FastCDC pads five zero bits into the mask value and changes the hash judgment statement to "*fp & mask == r*". If the masked bits of *fp* match a threshold value r, the current position will be declared as a chunk cut-point. Since Gear hash uses one left-shift and one addition operation to compute the rolling hash, this zero-padding scheme enables 10 bytes (i.e., B_i, ... , B_{i+9}), instead of the original five bytes, to be involved in the final

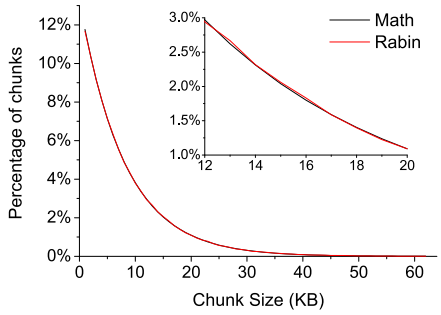


Figure 7: Chunk-size distribution of the Rabin-based CDC approach with average chunk size of 8KB and without the maximum and minimum chunk size requirements. "Rabin" and "Math" denote respectively our experimental observation and theoretical analysis (i.e., Equation (3)) of post-chunking chunk-size distribution, where they are shown to be nearly identical.

hash judgment by the five masked one bits (as the red box shown in Figure 6) and thus makes the sliding window size equal or similar to that of the Rabin-based CDC [25], minimizing the probability of the chunking position collision. As a result, FastCDC is able to achieve a deduplication ratio as high as that by the Rabin-based CDC.

Simplifying the hash judgment to accelerate CDC. The conventional hash judgment process, as used in the Rabin-based CDC, is expressed in the programming statement of "*fp mod D==r*" [25, 38]. For example, the Rabin-based CDC usually defines D and r as *0x02000* and *0x78*, according to the known open source project LBFS [25], to obtain the expected average chunk size of 8KB. In FastCDC, when combined with the zero-padding scheme introduced above and shown in Figure 6, the hash judgment statement can be optimized to "*fp* & *Mask*==0", which is equivalent to "*!fp* & *Mask*". Therefore, FastCDC's hash judgment statement reduces the register space for storing the threshold value r and avoids the unnecessary comparison operation that compares "*fp* & *Mask*" and r, thus further speeds up the CDC process as verified in Section 5.2.

4.3 Cut-point Skipping

Most of CDC-based deduplication systems impose a limit of the maximum and minimum chunk sizes, to avoid the pathological cases of generating many extremely large- or small-sized chunks by CDC [17, 19, 23–25, 29]. A common configuration of the average, minimum, and maximum parameters follows that used by LBFS [25], i.e., 8KB, 2KB, and 64KB. Our experimental observation and mathematical analysis suggest that the cumulative distribution of chunk size X in Rabin-based CDC approaches with an expected chunk size of *8 KB* (without the maximum and minimum chunk size requirements) follows an exponential distribution as follows:

$$P(X \leq x) = F(x) = (1 - e^{-\frac{x}{8192}}),\ x \geq 0. \quad (3)$$

Note that this theoretical exponential distribution in Equation 3 is based on the assumption that the data content and Rabin hashes of contents (recall Equation 1 and Figure 1 for CDC) follow a uniform distribution. Equation 3 suggests that the value of the expected chunk size will be 8KB according to the exponential distribution.

Figure 7 shows a comparison between the actual chunk-size distribution of the real-world datasets after the Rabin-based CDC and the chunk-size distribution obtained by the mathematical analysis based on Equation 3, which indicates that the two are almost identical. According to Equation 3, the chunks smaller than 2KB and larger than 64KB would account for about 22.12% and 0.03% of the total number of chunks respectively. This means that imposing the maximum chunk size requirement only slightly hurts the deduplication ratio but skipping cut-points before chunking to avoid generating chunks smaller than the prescribed minimum chunk size, or called *sub-minimum chunk cut-point skipping* , will impact the deduplication ratio significantly as evidenced in Figure 3. This is because a significant portion of the chunks are not divided truly according to the data contents, but forced by this cut-point skipping.

Given FastCDC's goal of maximizing the chunking speed, enlarging the minimum chunk size and skipping sub-minimum chunk cut-point will help FastCDC achieve a higher CDC speed by avoiding the operations for the hash calculation and judgment in the skipped region. This gain in speed, however, comes at the cost of reduced deduplication ratio. To address this problem, we will develop a normalized chunking approach, to be introduced in the next subsection.

It is worth noting that this cut-point skipping approach, by avoiding generating chunks smaller than the minimum chunk size, also helps increase the average generated chunk size. In fact, the average generated chunk size exceeds the expected chunk size by an amount equal to the minimum chunk size. This is because the F(x) in Equation 3 is changed to $(1 - e^{-\frac{x - MinSize}{8192}})$ after cut-point skipping, thus the value of the expected chunk size becomes 8KB + minimum chunk size, which will be verified in Section 5.3. The speedup achieved by skipping the sub-minimum chunk cut-point can be estimated by $1 + \frac{\textit{the minimum chunk size}}{\textit{the expected chunk size}}$. The increased chunking speed comes from the eliminated computation on the skipped region, which will also be verified in Section 5.3.

4.4 Normalized Chunking

In this subsection, we propose a novel chunking approach, called normalized chunking, to solve the problem of decreased deduplication ratio facing the cut-point

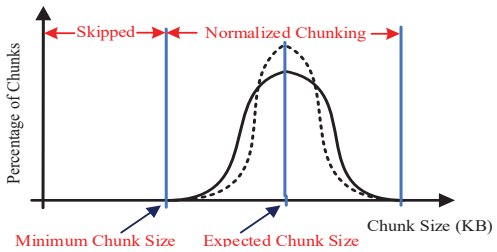


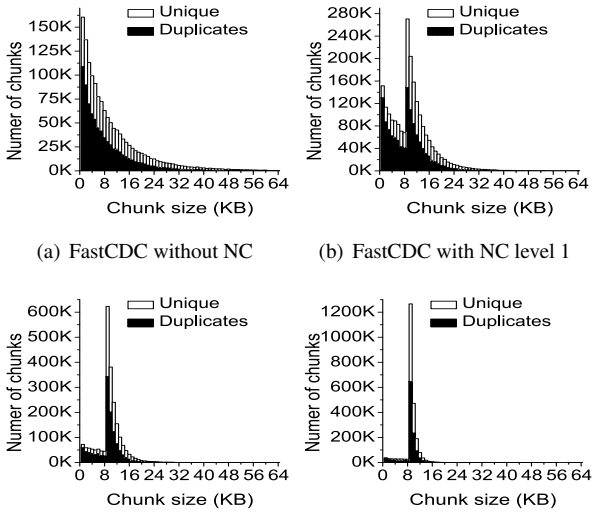
Figure 8: A conceptual diagram of the normalized chunking combined with the subminimum chunk cut-point skipping. The dotted line shows a higher level of normalized chunking.

skipping approach. As shown in Figure 8, normalized chunking generates chunks whose sizes are normalized to a specified region centered at the expected chunk size. After normalized chunking, there are almost no chunks of size smaller than the minimum chunk size, which means that normalized chunking enables skipping cut-points for subminimum chunks to reduce the unnecessary chunking computation and thus speed up CDC.

In our implementation of normalized chunking, we selectively change the number of effective mask bits (i.e., the number of '1' bits) for the hash-judging statement. For the traditional CDC approach with expected chunk size of 8KB (i.e., 2^{13}), 13 effective mask bits are used for hash judgment (e.g., *fp & 0x1fff==r*). For normalized chunking, more than 13 effective mask bits are used for hash judgment (e.g., *fp & 0x7fff==r*) when the current chunking position is smaller than 8KB, which makes it harder to generate chunks of size smaller than 8KB. On the other hand, fewer than 13 effective mask bits are used for hash judgment (e.g., *fp & 0x0fff==r*) when the current chunking position is larger than 8KB, which makes it easier to generate chunks of size larger than 8KB. Therefore, by changing the number of '1' bits in FastCDC, the chunk-size distribution will be approximately normalized to a specified region always larger than the minimum chunk size, instead of following the exponential distribution (see Figure 7).

Generally, there are three benefits or features of normalized chunking (NC):

- NC reduces the number of small-sized chunks, which makes it possible to combine it with the cut-point skipping approach to *achieve high chunking speed without sacrificing the deduplication ratio* as suggested in Figure 8.
- NC further improves the deduplication ratio by reducing the number of large-sized chunks, which compensates for the reduced deduplication ratio caused by reducing the number of small-sized chunks in FastCDC.
- The implementation of FastCDC does not add additional computing and comparing operations. It simply separates the hash judgment into two parts, before and after the expected chunk size.



(a) FastCDC without NC (b) FastCDC with NC level 1

(c) FastCDC with NC level 2 (d) FastCDC with NC level 3

Figure 9: Chunk-size distribution of FastCDC with normalized chunking (NC) at different normalization levels.

Figure 9 shows the chunk-size distribution after normalized chunking in comparison with FastCDC without NC on the TAR dataset (whose workload characteristics are detailed in Table 2 in Section 5.1). The normalization levels 1, 2, 3 indicate that the normalized chunking uses the mask bits of (14, 12), (15, 11), (16, 10), respectively, where the first and the second integers in the parentheses indicate the numbers of effective mask bits used in the hash judgment before and after the expected chunk size (or *normalized chunk size*) of 8KB . Figure 9 suggests that the chunk-size distribution is a reasonably close approximation of the normal distribution centered on 8KB at the normalization level of 2 or 3.

As shown in Figure 9, there are only a very small number of chunks smaller than 2KB or 4KB after normalized chunking while FastCDC without NC has a large number of chunks smaller than 2KB or 4KB (consistent with the discussion in Section 4.3). Thus, when combining NC with the cut-point skipping to speed up the CDC process, only a very small portion of chunk cut-points will be skipped in FastCDC, leading to nearly the same deduplication ratio as the conventional CDC approaches without the minimum chunk size requirement. In addition, normalized chunking allows us to enlarge the minimum chunk size to maximize the chunking speed without sacrificing the deduplication ratio.

It is worth noting that the chunk-size distribution shown in Figure 9 is not truly normal distribution but an approximation of it. Figures 9 (c) and (d) shows a closer approximation of normal distribution of chunk size achieved by using the normalization levels 2 and

Algorithm 1: FastCDC8KB

Input: data buffer, src; buffer length, n
Output: chunking breakpoint i

```
MaskS ← 0x0003590703530000LL;        // 15 '1' bits
MaskA ← 0x0000d90303530000LL;        // 13 '1' bits
MaskL ← 0x0000d90003530000LL;        // 11 '1' bits
MinSize ← 2KB;  MaxSize ← 64KB;
fp ← 0;  i ← MinSize;  NormalSize ← 8KB;
if n ≤ MinSize then
    return n;
if n ≥ MaxSize then
    n ← MaxSize;
else if n ≤ NormalSize then
    NormalSize ← n;
for ; i < NormalSize; i++; do
    fp = (fp << 1) + Gear[ src[i] ];
    if ! ( fp & MaskS ) then
        return i;            //if the masked bits are all '0'
for ; i < n; i++; do
    fp = (fp << 1) + Gear[ src[i] ];
    if ! ( fp & MaskL ) then
        return i;            //if the masked bits are all '0'
return i;
```

3. Interestingly, the highest normalization level of NC would be equivalent to Fixed-Size Chunking (FSC), i.e., all the chunk sizes are normalized to be equal to the expected chunk size. Since FSC has a very low deduplication ratio but extremely high chunking speed, it means that there will be a "sweet spot" among the normalization level, deduplication ratio, and chunking speed, which will be studied and evaluated in Section 5.

4.5 Putting It All Together

To put things together and in perspective. Algorithm 1 describes FastCDC combining the three key techniques: optimizing hash judgment, cut-point skipping, and normalized chunking (with the expected chunk size of 8KB). The data structure "Gear" is a predefined array of 256 random 64-bit integers with one-to-one mapping to the values of byte contents for chunking [38].

As shown in Algorithm 1, FastCDC uses normalized chunking to divide the chunking judgment into two loops with the optimized hash judgment. Note that FastCDC without normalized chunking is not shown here but can be easily implemented by using the new hash-judging statement "*! fp & MaskA*" where the *MaskA* is padded with 35 zero bits to enlarge the sliding window size to 48 bytes as that used in the Rabin-based CDC [25]. Note that *MaskA*, *MaskS*, and *MaskL* are three empirically derived values where the padded zero bits are almost evenly distributed for slightly higher deduplication ratio according to our large scale tests.

FastCDC implements normalized chunking by using mask value *MaskS* and *MaskL* to make the chunking judgment harder or easier (to generate chunks smaller or larger than the expected chunk size) when the current position is smaller or larger than the expected chunk size, respectively. And the number of '1' bits in *MaskS* and *MaskL* can be changed for different normalization levels. The minimum chunk size used in Algorithm 1 is 2*KB*, which can be enlarged to *4KB* or 8*KB* to further speed up the CDC process while combining with normalized chunking. Tuning the parameters of minimum chunk size and normalization level will be studied and evaluated in the next Section.

Name	Size	Workload descriptions
TAR	19 GB	85 tarred files from the open source projects such as GCC, GDB, Emacs, etc.
LNX	105 GB	260 versions of Linux source code files.
WEB	36 GB	15 days' snapshots of the website: *news.sina.com*, which are collected by crawling software *wget* with a maximum retrieval depth of 3.
VMA	117 GB	75 virtual machine images of different OS release versions, including CentOS, Fedora, Debian, etc.
VMB	1.9 TB	125 backups of an Ubuntu 12.04 virtual machine image in use by a research group.
RDB	1.1 TB	200 backups of the redis key-value store database.
SYN	1.4 TB	200 synthetic backups. The backup is simulated by the file create/delete/modify operations [31].

Table 2: Workload characteristics of the seven datasets used in the performance evaluation.

5 Performance Evaluation

5.1 Experimental Setup

Experimental Platform. To evaluate FastCDC, we implement a prototype of the data deduplication system on the Ubuntu 12.04.2 operating system running on a quad-core Intel i7-4770 processor at 3.4GHz, with a 16GB RAM. To better evaluate the chunking speed, another quad-core Intel i7-930 processor at 2.8GHz is also used for comparison.

Configurations for CDC and deduplication. Three CDC approaches, Rabin-, Gear-, and AE-based CDC, are used as the baselines for evaluating FastCDC. Rabin-based CDC is implemented based on the open-source project LBFS [25] (also used in many published studies [14, 22] or project [6]), where the sliding window size is configured to be 48 bytes. The Gear- and AE-based CDC schemes are implemented according to the algorithms described in their papers [38, 40], and we obtain performance results similar to and consistent with those reported in these papers. Here all the CDC approaches are configured with the maximum and minimum chunk sizes of 8× and $\frac{1}{4}$× of the expected chunk size, the same as configured in LBFS [25]. The deduplication prototype consists of approximately 3000 lines of C code, which is compiled by GCC 4.7.3 with the "-O3" compiler option.

Performance Metrics of Interest. ***Chunking speed***

Dataset	CDC	Expected Chunk Size of 4K (B)		Expected Chunk Size of 8K (B)		Expected Chunk Size of 16K (B)	
		Dedup Ratio	Avg. Chunk Size	Dedup Ratio	Avg. Chunk Size	Dedup Ratio	Avg. Chunk Size
TAR	RC	54.81%	5561	47.58%	11873	41.23%	24067
	GC	51.68% (-5.71%)	6094 (+9.58%)	44.90% (–5.64%)	12651 (+6.55%)	38.05% (–7.71%)	28743 (+19.4%)
	FC	54.14% (-1.22%)	5722 (+2.90%)	47.64% (+0.13%)	12192 (+2.69%)	41.26% (+0.08%)	24462 (+1.64%)
LNX	RC	97.69%	3828	97.25%	5978	96.80%	8188
	GC	97.78% (+0.09%)	3473 (–9.27%)	97.33% (+0.09%)	5644 (-5.59%)	96.88% (+0.08%)	7932 (-3.13%)
	FC	97.69% (+0.00%)	3845 (+0.44%)	97.26% (+0.01%)	5969 (-0.15%)	96.82% (+0.01%)	8176 (-0.15%)
WEB	RC	96.50%	5011	95.09%	9985	93.59%	19154
	GC	95.68% (-0.85%)	7091 (+41.5%)	94.09% (-1.13%)	17069 (+70.9%)	92.92% (-0.72%)	24960 (+30.3%)
	FC	96.14% (-0.38%)	5330 (+6.37%)	94.02% (-1.43%)	10725 (+7.41%)	93.21% (-0.40%)	19740 (+3.06%)
VMA	RC	42.99%	6367	38.23%	12743	32.97%	25485
	GC	42.60% (-0.91%)	5798 (-8.94%)	37.57% (-1.73%)	12069 (–5.29%)	32.23% (-2.13%)	24177 (–5.37%)
	FC	42.97% (-0.04%)	6293 (-1.16%)	37.96% (-0.72%)	12787 (+0.35%)	32.79% (-0.55%)	25620 (+0.53%)
VMB	RC	96.41%	5958	96.13%	11937	95.76%	24100
	GC	96.41% (+0.00%)	5662 (–4.96%)	96.06% (-0.07%)	11477 (–3.86%)	95.66% (-0.10%)	23260 (–3.49%)
	FC	96.39% (–0.02%)	6021 (+1.05%)	96.09% (-0.04%)	12138 (+1.68%)	95.70% (-0.06%)	24384 (+0.01%)
RDB	RC	97.36%	5116	95.53%	10232	92.05%	20479
	GC	97.20% (-0.16%)	5463 (+6.78%)	95.21% (-0.33%)	10923 (+6.75%)	91.49% (-0.60%)	21820 (+6.55%)
	FC	97.35% (-0.02%)	5118 (+0.04%)	95.50% (-0.03%)	10238 (+0.06%)	92.01% (-0.02%)	20479 (+0.00%)
SYN	RC	95.64%	5479	93.64%	10954	90.72%	21927
	GC	96.03% (+0.41%)	5338 (-2.57%)	94.30% (+0.70%)	10675 (-2.55%)	91.43% (+0.68%)	21325 (-2.75%)
	FC	95.65% (+0.01%)	5473 (-0.11%)	93.67% (+0.03%)	10945 (-0.08%)	90.73% (+0.02%)	21924 (-0.01%)

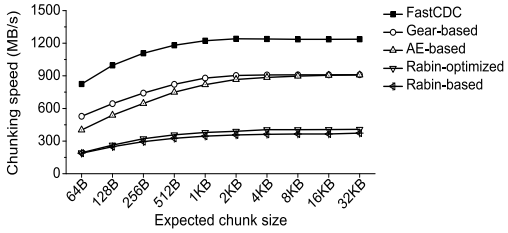
Table 3: A comparison among the Rabin-based CDC (RC), Gear-based CDC (GC), and FastCDC (FC) approaches in terms of the deduplication ratio and the average size of generated chunks, as a function of the expected chunk size.

is measured by the in-memory processing speed of the evaluated CDC approaches and obtained by the average speed of five runs. ***Deduplication ratio*** is measured in terms of the percentage of duplicates detected after CDC, i.e., $\frac{The\ size\ of\ duplicate\ data\ detected}{Total\ data\ size\ before\ deduplication}$. ***Average chunk size*** is $\frac{Total\ data\ size}{Number\ of\ chunks}$ after CDC, which reflects the metadata overhead for deduplication indexing.

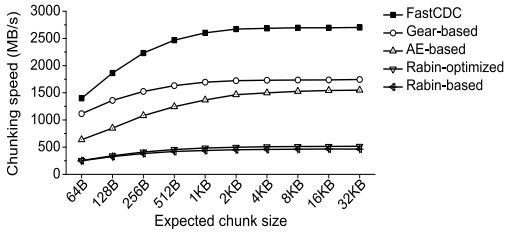
Evaluated Datasets. Seven datasets with a total size of about 5 TB are used for evaluation as shown in Table 2. These datasets consist of the various typical workloads of deduplication, including the source code files, virtual machine images, database snapshots, etc., whose deduplication ratios vary from 40% to 97%.

5.2 A Study of Optimizing Hash Judgment

This subsection discusses an empirical study of FastCDC using the optimized hash judgment. Figure 10 shows the chunking speed of the five CDC approaches running on the RDB dataset, as a function of the expected chunk size and all using the minimum chunk size of $\frac{1}{4}\times$ of that for cut-point skipping. The Rabin-optimized approach employs the technique of simplifying the hash judgment proposed in Section 4.2 but only achieves a little acceleration, this is because the hashing stage is the main bottleneck for Rabin-based CDC. In general, the Rabin-based CDC has the lowest speed, and the AE- and Gear-based CDC are about 3× faster than Rabin. For the AE-based CDC, its chunking speed is similar to that of the Gear-based CDC when the expected chunk size ranges from 2∼16 KB but is much lower than that of Gear when the expected chunk size is smaller than 1 KB. FastCDC is about 5× faster than Rabin and 1.5× faster than Gear



(a) Intel i7-930



(b) Intel i7-4770

Figure 10: Chunking speed, as a function of the expected chunk size, of Rabin-, Rabin (optimized)- Gear-, and AE-based CDC, and FastCDC on two CPU processors.

and AE regardless of the speed of the CPU processor and the expected chunk size. The high chunking speed of FastCDC stems from its simplification of the hash judgment after the fast Gear hash is used for chunking as described in Section 4.2.

Table 3 shows the deduplication ratio and the average size of generated chunks (post-chunking) achieved by the three CDC approaches. We compare the Gear-based CDC (GC), and FastCDC (FC) approaches against the classic Rabin-based CDC (i.e., the baseline) and record the percentage differences (in parentheses). AE-based CDC has nearly the same deduplication ratio as Rabin,

thus is not shown in this table due to space limit.

In general, FastCDC achieves nearly the same deduplication ratio as the Rabin-based CDC regardless of the expected chunk size and workload, and the difference between them is only about ±0.1∼1.4% as shown in the 3rd, 5th, 7th columns in Table 3. On the other hand, the Gear-based CDC has a much lower deduplication ratio on the datasets TAR, WEB, and VMA due to its limited sliding window size as discussed in Section 3.

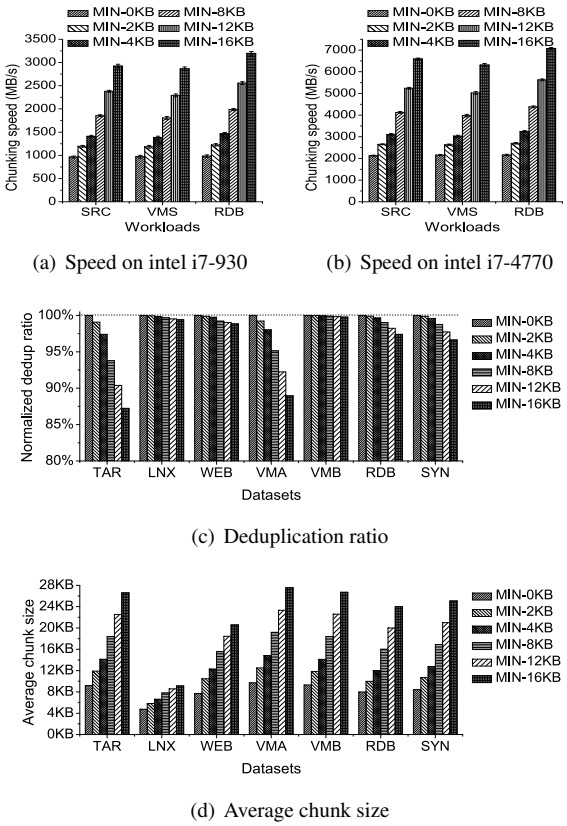
For the metric of the average size of generated chunks, the difference between the Rabin-based CDC and FastCDC is smaller than ±0.1% on most of the datasets. For the datasets TAR and WEB, FastCDC has 1∼7% larger average chunk size than Rabin-based CDC, which is acceptable since the larger average chunk size means fewer chunks and fingerprints for indexing in a deduplication system (without sacrificing deduplication ratio) [33]. But for the Gear-based CDC, the average chunk size differs significantly in some datasets while its deduplication ratio is still a bit lower than other CDC approaches due to its smaller sliding window size.

In summary, FastCDC achieves a chunking speed that is 5× higher than the Rabin-based CDC while satisfactorily solving the problem of low deduplication ratio facing the Gear-based CDC, as shown in Figure 10 and Table 3.

5.3 Evaluation of Cut-point Skipping

This subsection discusses the evaluation results of cut-point skipping technique. Figures 11 (a) and (b) show the impact of applying different minimum chunk sizes on the chunking speed of FastCDC. Since the chunking speed is not so sensitive to the workloads, we only show the three typical workloads in Figure 11. In general, cut-point skipping greatly accelerates the CDC process since the skipped region will not be hash-processed by CDC. The speedup of the FastCDC applying the minimum chunk sizes of 4KB and 2KB over the FastCDC without the constraint of the minimum chunk size (i.e., Min-0KB) is about 1.25× and 1.50× respectively, which is consistent with the equation $1+\frac{\textit{the minimum chunk size}}{\textit{the expected chunk size}}$ as discussed in Section 4.3.

Figures 11 (c) and (d) show the impact of applying different minimum chunk sizes on the deduplication ratio and average generated chunk size of FastCDC. In general, deduplication ratio declines with the increase of the minimum chunk size applied in FastCDC, but not proportionally. For the metric of the average generated chunk size in FastCDC, it is approximately equal to the summation of the expected chunk size and the applied minimum chunk size. This means that the MIN-4KB solution has the average chunk size of *8+4=12 KB*, leading to fewer chunks for fingerprints indexing in deduplication systems. Note that the increased portion of the average generated chunk size is not always equal to the size



(a) Speed on intel i7-930

(b) Speed on intel i7-4770

(c) Deduplication ratio

(d) Average chunk size

Figure 11: Chunking performance of FastCDC with the expected chunk size of 8KB but different minimum chunk sizes on two different CPU processors.

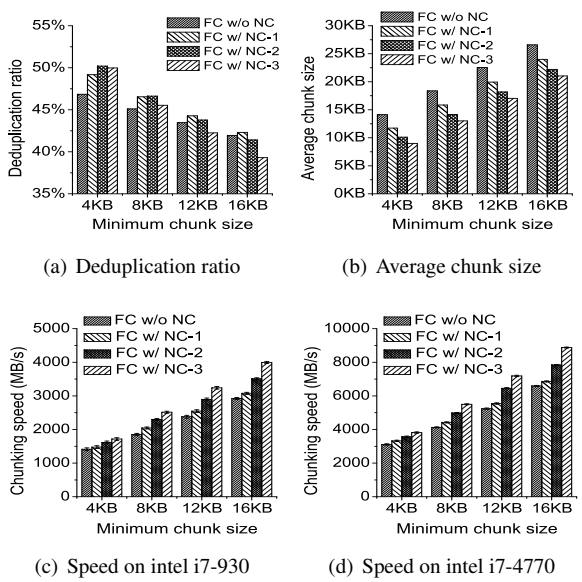
of the applied minimum chunk size, because the Rabin hashes of contents may not strictly follow the uniform distribution (as described in Equation 3 in Section 4.3) on some datasets.

In summary, the results shown in Figure 11 suggest that cut-point skipping helps obtain higher chunking speed and increase the average chunk size but at the cost of decreased deduplication ratio. The decreased deduplication ratio will be addressed by normalized chunking as evaluated in the next two subsections.

5.4 Evaluation of Normalized Chunking

In this subsection, we conduct a sensitivity study of normalized chunking (NC) on the TAR dataset, as shown in Figure 12. Here the expected chunk size of FastCDC without NC is 8KB and the normalized chunk size of FastCDC with NC is configured as the 4KB + minimum chunk size. The normalization levels 1, 2, 3 refer to the three pairs of numbers of effective mask bits (14, 12), (15, 11), (16, 10) respectively that normalized chunking applies when the chunking position is smaller or larger than the normalized (or expected) chunk size, as discussed in Section 4.4.

Figures 12 (a) and (b) suggest that normalized chunk-



(a) Deduplication ratio (b) Average chunk size

(c) Speed on intel i7-930 (d) Speed on intel i7-4770

Figure 12: Evaluation of comprehensive performance of normalized chunking with different normalization levels.

ing (NC) detects more duplicates when the minimum chunk size is about 4KB and 8KB but slightly reduces the average generated chunk size, in comparison with FastCDC without NC. This is because NC reduces the number of large-sized chunks as shown in Figure 9 and discussed in Section 4.4. The results also suggest that NC touches the "sweet spot" of deduplication ratio at the normalization level of 2 when the minimum chunk size is 4KB or 8KB. This is because the very high normalization levels tend to have a similar chunk-size distribution to the Fixed-Size Chunking as shown in Figure 9 in Section 4.4, which fails to address the *boundary-shift* problem and thus detects fewer duplicates. Figures 12 (c) and (d) suggest that NC, when combined with the approach of enlarging the minimum chunk size for cut-point skipping, greatly increases the chunking speed on the two tested processors. In addition, the average chunk sizes of datasets WEB and LNX are smaller than the minimum chunk size, which results from the many very small files whose sizes are much smaller than the minimum chunk size in the two datasets.

In general, considering the three metrics of chunking speed, average generated chunk size, and deduplication ratio as a whole, as shown in Figure 12, NC-2 with MinSize of 8KB maximizes the chunking speed without sacrificing the deduplication ratio. NC-2 with MinSize of 4KB achieves the highest deduplication ratio but with only a small acceleration of the chunking speed .

5.5 Comprehensive Evaluation of FastCDC

In this subsection, we comprehensively evaluate the performance of FastCDC with the combined capability of the three key techniques: optimizing hash judgment, cut-point skipping, and normalized chunking (using NC-2 as suggested by the last subsection). Four approaches are tested for evaluation: RC with Min2KB (or RC-MIN-2KB) is Rabin-based CDC used in LBFS [25]; FC with Min2KB (or FC-MIN-2KB) uses the techniques of optimizing hash judgment and cut-point skipping with a minimum chunk size of 2KB; FC-NC with Min4KB and FC-NC with Min8KB refer to FastCDC using all the three techniques with a minimum chunk size of 4KB and 8KB, respectively. To better evaluate the deduplication ratio, Fixed-Size Chunking (XC) is also tested using the average chunk size of 10KB.

Dataset	RC w/ Min2KB	FC w/ Min2KB	FC-NC w/ Min4KB	FC-NC w/ Min8KB	XC w/ 10KB
TAR	47.58%	47.64%	50.19%	47.18 %	12.21%
LNX	97.25%	97.26%	97.35%	97.10%	96.51%
WEB	95.09%	94.02%	95.47%	94.44%	93.19%
VMA	38.23%	37.96%	40.31%	38.15%	18.26%
VMB	96.13%	96.09%	96.24%	96.11%	95.68%
RDB	95.53%	95.50%	96.71%	95.70%	9.80%
SYN	93.64%	93.67%	94.09%	92.62%	75.06%

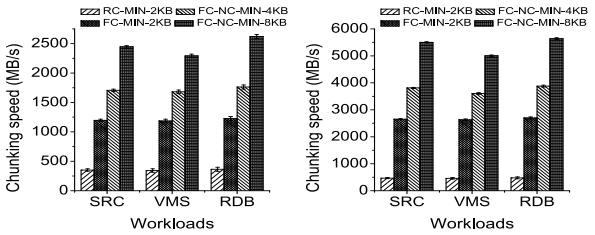
Table 4: Comparison of deduplication ratio achieved by the five chunking approaches. Note that "FC"and "FC-NC" refer to the full FastCDC without and with normalized chunking respectively, in this subsection.

Dataset	RC w/ Min2KB	FC w/ Min2KB	FC-NC w/ Min4KB	FC-NC w/ Min8KB	XC w/ 10KB
TAR	11873	12192	10347	14076	10240
LNX	5978	5969	6288	7585	6477
WEB	9985	10725	9327	12862	9513
VMA	12743	12787	11161	15031	10239
VMB	11937	12138	10850	15148	10239
RDB	10232	10238	9751	13846	10240
SYN	10954	10945	10318	14123	10240

Table 5: Average chunk size generated by the five chunking approaches on the seven datasets.

Evaluation results in Table 4 suggest that FC with Min2KB achieves nearly the same deduplication ratio as Rabin-based approach. FC-NC with Min4KB achieves the highest deduplication ratio among the five approaches while Fixed-Size Chunking (XC) has the lowest deduplication ratio. Note that XC works well on the LNX, WEB, VMB datasets, because LNX and WEB datasets have many files smaller than the fixed-size chunk of 10KB (and thus the average generated chunk size also smaller than 10KB) and VMB has many structured backup data (and thus VMB is suitable for XC).

Table 5 shows that RC and FC with Min2KB and XC generate similar average chunk size while FC-NC with Min4KB has a slightly small average chunk size. But the approach of FC-NC with Min8KB has a much smaller average chunk size, which means that it generates fewer chunks and thus less metadata for deduplication processing. Meanwhile, FC-NC with Min8KB still achieves a comparable deduplication ratio, slightly lower than RC as shown in Table 4, while providing a much higher



(a) Speed on intel i7-930 (b) Speed on intel i7-4770

Figure 13: Chunking speed of the four CDC approaches.

Approaches	Instructions	IPC	CPU cycles
RC-MIN-2KB	38, 829, 037	2.35	16, 537, 973
FC-MIN-2KB	15, 074, 950	4.37	3, 452, 146
FC-NC-MIN-4KB	11, 008, 372	4.82	2, 284, 453
FC-NC-MIN-8KB	7, 750, 124	4.82	1, 608, 033

Table 6: Number of instructions, instructions per cycle (IPC), and CPU cycles required to chunk 1MB data by the four CDC approaches on the Intel i7-4770 processor.

chunking speed as discussed later.

Figure 13 suggests that FC-NC with Min8KB has the highest chunking speed, about 10× faster than the Rabin-based approach, about 2× faster than FC with Min2KB. This is because FC-NC with Min8KB is the final FastCDC using all the three techniques to speed up the CDC process. In addition, FC-NC with Min4KB is also a good CDC candidate since it has the highest deduplication ratio while also working well on the other two metrics of chunking speed and the average generated chunk size. Note that XC is not shown here because it has almost no computation overhead for chunking.

Table 6 further studies the CPU overhead among the four CDC approaches. The CPU overhead is averaged on 1000 test runs by the Linux tool "Perf". The results suggest that FC-NC-MIN-8KB has the fewest instructions for CDC computation, the highest IPC (instructions per cycle), and thus the least CPU time overhead, i.e., CPU cycles. Generally, FastCDC greatly reduces the number of instructions for CDC computation by using the techniques of Gear-based hashing and optimizing hash judgment (i.e., "FC-MIN-2KB"), and then minimizes the number of computation instructions by enlarging the minimum chunk size for cut-point skipping and combining normalized chunking (i.e., "FC-NC-MIN-8KB"). In addition, FastCDC increases the IPC for the CDC computation by well pipelining the instructions of hashing and hash-judging tasks in up-to-date processors. Therefore, these results explain why FastCDC is about 10× faster than Rabin-based CDC is that the former not only reduces the number of instructions, but also increases the IPC for the CDC process.

In summary, as shown in Tables 4, 5, 6 and Figure 13, FastCDC (i.e., FC-NC-MIN-8KB) significantly speeds up the chunking process and achieves a comparable deduplication ratio while reducing the number of generated chunks by using a combination of the three key techniques proposed in Section 4.

6 Conclusion and Future Work

In this paper, we propose FastCDC, a much faster CDC approach for data deduplication than the state-of-the-art CDC approaches while achieving a comparable deduplication ratio. The main idea behind FastCDC is the combined use of three key techniques, namely, optimizing the hash judgment for chunking, subminimum chunk cut-point skipping, and normalized chunking. Our experimental evaluation demonstrates that FastCDC obtains a chunking speed that is about 10× higher than that of the Rabin-based CDC and about 3× that of the Gear- and AE-based CDC while achieving nearly the same deduplication ratio as the Rabin-based CDC.

In our future work, we plan to incorporate FastCDC in some other deduplication systems that are sensitive to the CPU overhead of content-defined chunking, such as QuickSync [9], to further explore the potentials and benefits of FastCDC. We also plan to release the FastCDC source code to be shared with the deduplication and storage systems research community.

Acknowledgments

We are grateful to our shepherd Scott Rixner and the anonymous reviewers for their insightful comments and feedback. The work was partly supported by NSFC No. 61502190, 61502191, 61232004, and 61402061; 863 Project 2013AA013203; State Key Laboratory of Computer Architecture, No. CARCH201505; Fundamental Research Funds for the Central Universities, HUST, under Grant No. 2015MS073; US NSF under Grants CNS-1116606 and CNS-1016609, Key Laboratory of Information Storage System, Ministry of Education, China, and Sangfor Technologies Co., Ltd.

References

[1] AGGARWAL, B., AKELLA, A., ANAND, A., ET AL. EndRE: an end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation (NSDI'10)* (San Jose, CA, USA, April 2010), USENIX Association, pp. 14–28.

[2] AL-KISWANY, S., GHARAIBEH, A., SANTOS-NETO, E., ET AL. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th international symposium on High Performance Distributed Computing (HPDC'08)* (Boston, MA, USA, June 2008), ACM Association, pp. 165–174.

[3] ANAND, A., MUTHUKRISHNAN, C., AKELLA, A., AND RAMJEE, R. Redundancy in network traffic: findings and

implications. In *Proceedings of the 11th international Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS-Performance 2009)* (Seattle, WA, USA, June 2009), ACM Association, pp. 37–48.

[4] BHAGWAT, D., POLLACK, K., LONG, D. D., SCHWARZ, T., MILLER, E. L., AND PÂRIS, J.-F. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of The 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'06)* (Monterey, CA, USA, September 2006), IEEE Computer Society Press, pp. 413–421.

[5] BHATOTIA, P., RODRIGUES, R., AND VERMA, A. Shredder: GPU-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, USA, February 2012), USENIX Association, pp. 1–15.

[6] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT'08)* (Toronto, Canada, October 2008), ACM, pp. 72–81.

[7] BJØRNER, N., BLASS, A., AND GUREVICH, Y. Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences 76*, 3 (2010), 154–203.

[8] BRODER, A. Some applications of Rabin's fingerprinting method. *Sequences II: Methods in Communications, Security, and Computer Science* (1993), 1–10.

[9] CUI, Y., LAI, Z., WANG, X., DAI, N., AND MIAO, C. QuickSync: Improving Synchronization Efficiency for Mobile Cloud Storage Services. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking* (Paris, France, Sept. 2015), ACM, pp. 592–603.

[10] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIX'10)* (Boston, MA, USA, June 2010), USENIX Association, pp. 1–14.

[11] DUBNICKI, C., KRUUS, E., LICHOTA, K., AND UNGUREANU, C. Methods and systems for data management using multiple selection criteria, Dec. 1 2006. US Patent App. 11/566,122.

[12] EL-SHIMI, A., KALACH, R., KUMAR, A., ET AL. Primary data deduplication–large scale study and system design. In *Proceedings of the 2012 conference on USENIX Annual Technical Conference (USENIX'12)* (Boston, MA, USA, June 2012), USENIX Association, pp. 1–12.

[13] ESHGHI, K., AND TANG, H. K. A framework for analyzing and improving content-based chunking algorithms. *Tech. Rep. HPL-2005-30(R.1),Hewlett Packard Laboratories, Palo Alto* (2005).

[14] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., ZHANG, Y., AND TAN, Y. Design tradeoffs for data deduplication performance in backup workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)* (Santa Clara, CA, USA, February 2015), USENIX Association, pp. 331–344.

[15] GHARAIBEH, A., AL-KISWANY, S., GOPALAKRISHNAN, S., ET AL. A GPU accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)* (Chicago, Illinois, USA, June 2010), ACM Association, pp. 167–178.

[16] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX conference on USENIX Annual Technical Conference (USENIX'11)* (Portland, OR, USA, June 2011), USENIX Association, pp. 1–14.

[17] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal content defined chunking for backup streams. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'10)* (San Jose, CA, USA, February 2010), USENIX Association, pp. 1–14.

[18] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)* (San Jose, CA, USA, February 2013), USENIX Association, pp. 183–197.

[19] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., ET AL. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)* (San Jose, CA, February 2009), vol. 9, USENIX Association, pp. 111–123.

[20] LILLIBRIDGE, M. D. Parallel processing of input data to locate landmarks for chunks, Aug. 16 2011. US Patent 8,001,273.

[21] LU, G., JIN, Y., AND DU, D. H. Frequency based chunking for data de-duplication. In *Proceedings of 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS'10)* (Miami Beach, FL, USA, August 2010), IEEE Computer Society Press, pp. 287–296.

[22] MEISTER, D., KAISER, J., BRINKMANN, A., ET AL. A study on data deduplication in HPC storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'02)* (Salt Lake City, Utah, USA, June 2012), IEEE Computer Society Press, pp. 1–11.

[23] MEYER, D., AND BOLOSKY, W. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'11)* (San Jose, CA, USA, February 2011), USENIX Association, pp. 229–241.

[24] MIN, J., YOON, D., AND WON, Y. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers 60*, 6 (2011), 824–840.

[25] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'01)* (Banff, Canada, October 2001), ACM Association, pp. 1–14.

[26] POLICRONIADES, C., AND PRATT, I. Alternatives for detecting redundancy in storage systems data. In *Proceedings of USENIX Annual Technical Conference, General Track* (Boston, MA, USA, June 2004), USENIX Association, pp. 73–86.

[27] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'02)* (Monterey, CA, USA, January 2002), USENIX Association, pp. 1–13.

[28] RABIN, M. O. *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[29] ROMAŃSKI, B., HELDT, Ł., KILIAN, W., LICHOTA, K., AND DUBNICKI, C. Anchor-driven subchunk deduplication. In *Proceedings of The 4th Annual International Systems and Storage Conference (SYSTOR'11)* (Haifa, Israel, May 2011), ACM Association, pp. 1–13.

[30] SHILANE, P., HUANG, M., WALLACE, G., ET AL. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, USA, February 2012), USENIX Association, pp. 1–14.

[31] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX'12)* (Boston, MA, USA, June 2012), USENIX Association, pp. 24–34.

[32] TEODOSIU, D., BJORNER, N., GUREVICH, Y., MANASSE, M., AND PORKKA, J. Optimizing file replication over limited bandwidth networks using remote differential compression. *Microsoft Research TR-2006-157* (2006).

[33] WALLACE, G., DOUGLIS, F., QIAN, H., ET AL. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST'12)* (San Jose, CA, February 2012), USENIX Association, pp. 1–14.

[34] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference (USENIX'11)* (Portland, OR, USA, June 2011), USENIX Association, pp. 285–298.

[35] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Similarity and locality based indexing for high performance data deduplication. *IEEE Transactions on Computers 64*, 4 (2015), 1162–1176.

[36] XIA, W., JIANG, H., FENG, D., AND TIAN, L. Accelerating data deduplication by exploiting pipelining and parallelism with multicore or manycore processors. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12 Poster)* (San Jose, CA, USA, February 2012), USENIX Association, pp. 1–2.

[37] XIA, W., JIANG, H., FENG, D., TIAN, L., FU, M., AND WANG, Z. P-dedupe: Exploiting parallelism in data deduplication system. In *Proceedings of the 7th International Conference onNetworking, Architecture and Storage (NAS'12)* (Xiamen, China, June 2012), IEEE Computer Society Press, pp. 338–347.

[38] XIA, W., JIANG, H., FENG, D., TIAN, L., FU, M., AND ZHOU, Y. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation 79* (2014), 258–272.

[39] YU, C., ZHANG, C., MAO, Y., AND LI, F. Leap-based content defined chunking—theory and implementation. In *Proceedings of the 31th Symposium on Mass Storage Systems and Technologies (MSST'15)* (Santa Clara, CA, USA, June 2015), IEEE, pp. 1–12.

[40] ZHANG, Y., JIANG, H., FENG, D., XIA, W., FU, M., HUANG, F., AND ZHOU, Y. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *Proceedings of IEEE INFOCOM 2015* (Hongkong, April 2015), IEEE Computer Society Press, pp. 1337–1345.

[41] ZHOU, Y., FENG, D., XIA, W., FU, M., HUANG, F., ZHANG, Y., AND LI, C. SecDep: A User-Aware Efficient Fine-Grained Secure Deduplication Scheme with Multi-Level Key Management. In *Proceedings of IEEE 31th Symposium on Mass Storage Systems and Technologies (MSST'15)* (Santa Clara, CA, USA, June 2015), IEEE, pp. 1–12.

[42] ZHU, B., LI, K., AND PATTERSON, R. H. Avoiding the disk bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)* (San Jose, CA, USA, February 2008), vol. 8, USENIX Association, pp. 1–14.

Unsafe Time Handling in Smartphones

Abhilash Jindal
Purdue University

Prahlad Joshi
Purdue University

Y. Charlie Hu
Purdue University

Samuel Midkiff
Purdue University

Abstract

Time manipulation, typically done using gettime() and settime(), happens extensively across all software layers in smartphones, from the kernel, to the framework, to millions of apps. This paper presents the first study of a new class of software bugs on smartphones called sleep-induced time bugs (SITB). SITB happens when the phone is suspended, due to the aggressive sleeping policy adopted in smartphones, in the middle of a time critical section where time is being manipulated and delay caused by unexpected phone suspension alters the intended program behavior.

We first characterize time usages in the Android kernel, framework, and 978 apps into four categories and study their vulnerabilities to system suspension. Our study shows time manipulation happens extensively in all three software layers, totaling 1047, 1737 and 7798 times, respectively, and all four usage patterns are vulnerable to SITBs. We then present a tool called KLOCK, that makes use of a set of static analyses to systematically identify sleep-induced time bugs in three of the four time usage categories. When applied to five different Android Linux kernels, KLOCK correctly flagged 63 SITB-vulnerable time manipulation instances as time bugs.

1 Introduction

Smartphones designs are increasingly subject to three diametrically opposed constraints: phones must have increasing software and hardware functionality which can increase power requirements; phones have limited form factor and weight which bounds the size of their battery and therefore their power supply; and phones must have ever increasing battery life to meet user expectations and be competitive in the market place. The push for maximal energy savings under these constraints quickly drove their OSes, such as Android, to pursue an aggressive system sleeping policy. After some set period of user inactivity, *i.e.,* the user has not touched the screen or any peripheral buttons on the device, the OS power management triggers the phone's system on chip (SOC) to enter its default state, *system suspend*, where all components on the SOC are suspended, RAM is put in a self-refresh mode, and the SOC drains close-to-zero battery power.

The difficulty with this approach is that applications, the application framework (supplied by the OS vendor and providing low-level services to the apps) and the kernel (which implements many of the low-level services provided by the framework) often have *time critical sections* that are not part of interactive code. A time critical section is a dynamic code region (*i.e.,* code that may not be textually contiguous but is logically related) over which the system should not suspend.

To prevent the system from suspending when performing time critical work, smartphone OSes have exported mechanisms to allow programmers to prevent system suspension in selected parts of the program. Primary among these mechanisms is the *wakelock*, which, when acquired, prevents the system from suspending and when released allows it to suspend if nothing else is preventing the suspension.

However, mobile phone kernels and apps are complex. They utilize an event driven programming model and are often concurrent. Combining this complicated programming model with explicitly managing the SOC suspend/awake cycle unavoidably results in *sleep disorder bugs*, *i.e.,* programming mistakes in system suspension management that result in unintended app or hardware device behavior.

Pathak *et al*. [25] presented the first study of sleep disorder bugs, focusing on a class of bugs that result from not releasing wakelocks in apps, preventing the phone from going to sleep and draining energy. Jindal *et al*. [16] studied sleep conflicts, another class of sleep bugs that happen in device drivers where a phone's component (*e.g.,* WiFi) is left in a high power state wasting battery. Sleep conflicts occur when a component in high power state is unable to transition back to its base power

state because the system got suspended before the device driver responsible for driving the power transition could run. In both studies, the sleep bugs targeted do not lead to incorrect program behavior; instead they cause excessive battery drain due to preventing the SOC/CPU or I/O devices from going to sleep.

This paper studies a new class of sleep-related bugs, called *sleep-induced time bugs*. Sleep-induced time bugs can occur at all levels of the mobile phone ecosystem – in apps, the framework and the OS code. Unlike previously studied sleep bugs which lead to energy leaks, sleep-induced time bugs manifest as logical errors resulting from time values becoming "stale" because the CPU sleeps during the manipulation of time related values. The manipulations occur via `gettime()`, `settime()`, time arithmetic APIs, and simple arithmetic operations, and the new class of bugs manifest themselves as incorrect values in program variables rather than an incorrect power state of the device hardware.

Sleep-induced time bugs are difficult to reproduce and debug since they only arise during some particular, intricate timing between time-critical section execution and CPU suspension. Indeed when we submitted a patch to the bug we found in the DHT11 humidity and temperature sensor driver, the kernel maintainer responded [8] by saying:

> "I think it will fix an odd issue I have seen in a log file (apparently was completely off track debugging it). As this very likely is a real world issue, I'd recommend applying the patch to the fixes branch [sic]."

In this paper, we make three contributions towards understanding and treating sleep-induced time bugs.

First, we characterize time manipulation usages and their vulnerabilities to system suspension in the Android kernel, framework, and 978 apps. We find time manipulation happens extensively in all three software layers, totaling 1047, 1737 and 7798 times, respectively. We further classify all time usages into four categories: timed callbacks, setting the time, time arithmetic, and timed logging. The categorization uncovers the time critical sections in each category, their vulnerability to sleep-induced time bugs, as well their syntactic characteristics which give hints for detecting them.

Second, to allow programmers to isolate and fix sleep-induced time bugs, we present a tool called KLOCK that detects all instances of the first three categories of time manipulations in the Linux kernel. KLOCK exploits a key observation that the start and/or end of time critical sections due to time usage in the three categories are marked with a handful of APIs that get/set time or register timer callbacks in the Linux kernel. KLOCK makes use of a collection of sophisticated compiler analyses – Use-Def and Def-Use chains, points-to analysis and reaching definitions analysis as building blocks, and customizes their integration to detect time bugs in each of the three categories of time usages.

Third, we have implemented KLOCK and applied it to five Android Linux kernel versions. KLOCK aided in detecting 63 time bugs, out of which, we found 14 have been fixed and 7 files with bugs have been removed from later versions of the kernel. We reported the remaining 42 bugs to the corresponding Linux kernel mailing lists, and out of the 7 developers who have replied so far, all confirmed the reported bugs and accepted our patches.

Although we focus on the Android Linux kernel in this paper, we believe KLOCK is quite general and its design can be applied to detect sleep-induced time bugs in the framework, the apps, and other software systems that are vulnerable to system suspension, the de facto technique for energy saving on mobile systems.

2 Background

We start with a brief overview of the system suspension process aggressively triggered on modern smartphones and the complex set of clock options provided by the Linux kernel.

2.1 System Suspension in Smartphones

A modern smartphone consists of the SOC and numerous hardware I/O devices such as LCD, SD Card, WiFi NIC, cellular, GPS, cameras, and accelerometer. The SOC consists of the CPU, RAM, ROM, and micro-controller circuits for various phone devices such as GPS, graphics, video and audio. The default SOC power state is suspend, where all components on the SOC are suspended, RAM is put in a self-refresh mode, and the SOC drains close-to-zero battery power.

Wakelocks are a type of explicit power control APIs with two associated API calls, acquire and release. Wakelocks are also exported to the user space to support background services as well as non-interactive foreground jobs. The Android framework, apps, and device drivers extensively use wakelocks to ensure continuous execution of code sections.

When the last wakelock is released, the wakelock kernel module immediately attempts system suspension, by calling `pm_suspend()` to perform four tasks serially. First, the filesystem is synced by moving the buffered data from RAM to NAND. Second, all the user processes and kernel threads are frozen. Third, it attempts to suspend devices by calling the list of *suspend callbacks* registered by device drivers which power down their respective devices. Note that any suspend callback may return failure because it is waiting on a condition variable which

is set to false elsewhere in the kernel, which would abort the entire suspend process. Finally, all the CPU cores are disabled by calling the architecture specific code to complete the suspension.

Note that system suspension is only attempted when the last wakelock is released. If interrupts in the system are disabled, the running process cannot be context switched to another process that might release the wakelock or get interrupted by wakelock timer expiration. Thus disabling interrupts in a code section effectively prevents suspension.

In summary, system suspension will not succeed while a piece of code is executing if it (1) holds a wakelock; (2) disables interrupts indirectly preventing the last wakelock from getting released; or (3) sets a condition variable that causes a suspend callback to return failure and hence abort any suspension attempt.

2.2 Timekeeping in Linux

The Linux timekeeping subsystem is responsible for maintaining and providing current time to the rest of the kernel. The POSIX standard requires the timekeeping subsystem to maintain `CLOCK_REALTIME` which is the time elapsed since the midnight of January 1, 1970. `CLOCK_REALTIME` is first read from the real time clock during the kernel initialization phase and then later updated at every tick.

However, `CLOCK_REALTIME` is susceptible to sudden changes due to the user setting the time or from ntpd, making it particularly unsuitable to measure elapsed time of a code section. To overcome this, the POSIX standard mandates the timekeeping subsystem to provide `CLOCK_BOOTTIME` which gives the time elapsed since the boot time. `CLOCK_BOOTTIME` can not be set by the user or by ntpd and hence does not suffer from sudden discontinuities like `CLOCK_REALTIME`.

But `CLOCK_BOOTTIME` is not quite suitable for measuring code execution time, because it includes the time elapsed even while the SOC is suspended. For this reason, the POSIX standard introduced `CLOCK_MONOTONIC` which works like `CLOCK_BOOTTIME` but pauses during SOC suspension making it suitable for measuring program execution time.

Although `CLOCK_BOOTTIME` and `CLOCK_MONOTONIC` will not be reset to suddenly jump backward or forward, their rate is still adjusted slightly to fix clock drifts which is done autonomously by the timekeeping subsystem. For this reason, in addition to POSIX standards, the Linux timekeeping subsystem also provides `CLOCK_MONOTONIC_RAW` which is simply the local oscillator not disciplined by NTP, for use in cases where more accurate time is needed over very short intervals.

In summary, the myriad of clocks available in the Linux kernel and their subtle semantics pose a significant challenge to the developers, and using the wrong clock leads to vulnerabilities to unexpected events such as system suspension.

Listing 1: Sleep induced time bug in Linux kernel memcpy benchmark: system suspend can alter the time arithmetic result.

```
1 double do_memcpy_gettimeofday(memcpy_t fn, size_t
      len...) {
2   struct timeval tv_start, tv_end, tv_diff;
3   alloc_mem(&src, &dst, len);
4   gettimeofday(&tv_start, NULL);
5   fn(dst, src, len);
6   gettimeofday(&tv_end, NULL);
7   timersub(&tv_end, &tv_start, &tv_diff);
8   return len / timeval2double(&tv_diff);
9 }
```

3 Sleep-Induced Time Bugs

Time manipulation occurs frequently across all layers of smartphone software, from the kernel, to the framework, to the apps. Two factors together give rise to *sleep-induced time bugs* (SITB). First, the smartphone OS employs an aggressive system suspend policy. Second, time manipulation in smartphone software forms a *time critical section* (TICS) whose start and end are marked by time manipulation APIs or operations involving values obtained from the time manipulation APIs. Any delay within the TICS caused by the smartphone suspension will alter the intended program behavior and give rise to an SITB. More formally, an SITB happens when the smartphone CPU/SOC is suspended in the middle of a TICS that alters the intended program behavior. We discuss the impact of these bugs in Section 4.

We now illustrate a sleep-induced time bug in the Linux kernel memcpy benchmark, `/tools/perf/bench/mem-memcpy.c`. The benchmark code measures how much time each of the various memcpy functions takes to copy a single byte. A code snippet is shown in Listing 1. The function accepts a pointer to the function `fn` being benchmarked and the length of the memory block to be copied. Before calling `fn` to start copying in Line 5, the current time is read into variable `tv_start`. After `fn` returns, the current time is read in variable `tv_end` in Line 6. Line 7 computes the time taken by `fn` by computing the difference between `tv_start` and `tv_end`. Line 8 then calculates the rate of copying by dividing `len` by `time_diff`.

Consider the scenario where the CPU sleeps in between the two calls of `gettimeofday`, in or outside `fn`, `tv_start` is set to T1 and `tv_end` is set to T4, but the system was suspended from T2 until T3. The code will incorrectly compute the time taken by `fn` as (T4 - T1), while the actual time taken is (T2 - T1) + (T4 - T3), and return an erroneous copying rate.

Table 1: Time usage in the Anroid kernel, framework, and 978 apps.

Usage Pattern	Static Use Count			Example Usage in Kernel
	Kernel	Android	App	
Timer Callback	477	215	352	kernel/time/alarmtimer, fs/timerfd.c
Time Setting	17	8	1	kernel/time.c, drivers/rtc/alarm.c
Time Arithmetic (lower bound)	125	522	236	net/ipv4/tcp_probe.c, kernel/time/tick-sched.c drivers/cpuidle/cpuidle.c fs/jbd/transaction.c
Logging (upper bound)	453	992	7209	fs/dlm/ast.c, net/ wireless/mwifiex/cmdevt.c net/sunrpc/svcsock.c
Total	1072	1737	7798	

4 Characterizing Time Usage and Vulnerability to Sleep-induced Time Bugs

To understand the prevalence of time usage, typical time usage patterns, and their vulnerability to sleep-induced time bugs in smartphones, we examined and classified all the time usage in the Android kernel, the framework, and a set of 978 apps. The classification gave us many insights into the root causes of SITBs and hints on how to detect them.

4.1 Time Usage in the Android Ecosystem

As discussed in §2.2, the Linux timekeeping subsystem provides a myriad of different clocks and exports the APIs (except for `clock_monotonic_raw`) at every software layer, from device drivers all the way up to apps. We first read the API documentation to collect the list of such time APIs exposed at each software layer [6, 5, 4, 1, 2]. We then grepped for all the usages of respective APIs in the source code of the kernel, the Android framework, and a set of 978 apps which included the 100 most popular apps on Google Play which we manually downloaded and 878 apps we crawled the day before Android Market was switched to Google play. The app source code were obtained by decompiling the apk files using ded [3].

Table 1 shows that time manipulation is prevalent in the Android ecosystem, totalling 1072, 1737 and 7798 times in the Android kernel, framework, and 978 apps.

4.2 Categorizing Time Usage and Vulnerability

To understand the purposes of time manipulation widely used in the smartphone software layers, we manually inspected 50 time usages found in each software layer and found them to fall into the following four categories. Understanding the usage of each category in turn allows us to automatically search for all the instances of each usage, as explained below.

Listing 2: Code that generates waveform using timed callback.

```
1  // Generates a sawtooth wave on channel 0,
       square wave on channel 1
2  static void waveform_ai_interrupt(unsigned long
       arg) {
3    do_gettimeofday(&now);
4    elapsed_time = USEC_PER_SEC*(now.tv_sec-
       devpriv->last.tv_sec)+(now.tv_usec-
       devpriv->last.tv_usec);
5    devpriv->last = now;
6    num_scans = (devpriv->usec_remainder +
       elapsed_time) / devpriv->scan_period;
7    for (i = 0; i < num_scans; i++) {
8      sample = fake_waveform(dev, ...);
9      cfc_write_to_buffer(dev->read_subdev,sample
         );
10   }
11   devpriv->usec_current += elapsed_time;
12   mod_timer(&devpriv->timer, jiffies + 1);
13 }
14 static int waveform_attach(struct comedi_device
       *dev, struct comedi_devconfig *it) {
15   ..
16   init_timer(&(devpriv->timer));
17   devpriv->timer.function =
       waveform_ai_interrupt;
18 }
```

4.2.1 Case 1: Timed Callback

Usage Pattern. In this category, the code wishes to perform a certain task at a future time. The code registers an alarm with the system specifying the function that should be called and the time interval after which the callback should happen.

Listing 2 shows an example of how timed callbacks are set up and used in the kernel. The code generates waveforms of preconfigured shape. At driver initialization, function `waveform_attach` (line 14) is called which sets the pointer of the timed callback function `devpriv->timer.function`. The function `waveform_ai_interrupt` generates one period wide wave (lines 7-10) then recursively invokes itself via a timer callback (line 12).

Vulnerability. In this category of time usage, the duration from timer registration till the timeout (*i.e.*, when callback is supposed to be invoked) forms a time critical section. A time bug can arise when the CPU suspension happens in the middle of the TICS which delays the callback until the next time the CPU wakes up.

The waveform generation code in listing 2 contains a time bug. Since the driver does not protect against SOC suspension, the SOC might suspend after the timer is set at line 12 and cause large gaps in the waveform distorting its shape.

However, we observed not all such delays give rise to time bugs: time bugs arise only when the callback execution interacts with a peripheral I/O components. For

Listing 3: Clock synchronization in GsmServiceStateTracker.java class in Android (simplified for illustration).

```
1  wakelock.acquire();      //Keep the CPU on
2  x = gettime();           //Obtain external time
3  if (a condition){     //not based on x
4    /* lots of code */
5    x = f(x, z);         //Fix x using z
6    /* more code */
7  } else {
8    /* some code */
9    wakelock.release(); //Release wakelock
10   /* lots of code */
11   wakelock.acquire(); //Re-acquire wakelock
12 }
13 settime(x);             //Set hardware time
14 wakelock.release();     //Release wakelock
```

example, an alarm app that wishes to ring an alarm at a user-specified time must ensure that the callback is processed at the intended time. On the other hand, the process scheduler in the kernel also registers a timer callback in order to schedule a new process at the end of a time slice, but even if the CPU suspends in the middle, the delay in callback execution will have no impact on the scheduler semantics.

Occurrences. To count the occurrences of this type of time usage, we compiled a list of all callback registration APIs that are exported at each software layer by reading the documentation [6, 2], and counted the number of occurrences of those APIs in the source code. Table 1 shows that the time callback is widely used, for a total of 477, 215, and 352 times in the Linux kernel, the framework, and the 978 apps, respectively.

4.2.2 Case 2: Time Setting

Usage Pattern. In this category, the subject code updates the current system time. For example, code listing 3 shows an excerpt from GsmServiceStateTracker in the Android framework that obtains the external time (line 2), performs manipulation over it (lines 3-12), and sets the local time (line 13).

Vulnerability. In this category, the duration from gettime() to settime() forms a time critical section. A time bug will arise when the CPU suspends in the middle of the TICS which causes the new hardware time set to be incorrect. For example, code listing 3 sets the time (line 13) obtained from the network (line 2). But at line 9, the programmer mistakenly releases the wakelock, giving an opportunity for the CPU to suspend between line 9 and line 11. When this happens, line 13 will run after the next CPU wakeup, setting a stale time.

Occurrences. To count the occurrences of this type of time usage, we compiled a list of exported APIs at the three software layers for setting the current system time and searched for them in the source code. We found 17, 8, and 1 instances of this type of time usage in the kernel, framework and the 978 apps, respectively, as shown in Table 1.

Listing 4: Speed calculation in SpeedTest.net which measures the network connection speed.

```
1 protected Integer doInBackground(URL[]  r1) {
2   mStartTime = SystemClock.uptimeMillis();
3   //upload data
4 }
5 protected int getProgress(int  i0) {
6   //compute speed
7   i33 = i0 / (SystemClock.uptimeMillis() -
        mStartTime) / 1000;
8 }
```

4.2.3 Case 3: Time Arithmetic

Usage Pattern. Another common time usage pattern is to collect two timestamps and perform arithmetic over them. The arithmetic is performed either directly via integer or long arithmetic, or using Linux provided helper functions dedicated to performing time arithmetic, *e.g.*, `timespec_sub`.

Code listing 4 is extracted from the SpeedTest.net app [9] which measures the speed of network connection by registering two callback functions with the framework. The first callback takes a timestamp and saves it in `mStartTime` (line 2), and uploads data to the test server (line 3). The second callback then computes the speed by finding the elapsed time by subtracting `mStartTime` from the current time (line 7).

Vulnerability. In this category, the duration between the actions of getting two timestamps forms a time critical section. Time arithmetic programs are vulnerable to two kinds of vulnerability.

(a) Due to system suspension. A time bug will arise when the CPU suspends in the middle of the TICS and the time arithmetic will output an incorrect value. For example, in code listing 4, the TICS which starts at line 2 and ends at line 7 is not protected by any wakelock. As a result, the CPU may suspend before line 7, and the computed speed will be much lower than the actual one.

(b) Due to resetting the time. Time bugs will also arise in time arithmetic when the user or ntpd resets the current time between the actions of obtaining two timestamps.[1] In addition to causing elongated elapsed time (as in system suspension vulnerability (a)), this vulnerability can cause elapsed time to elongate or shrink or even become negative since the time can be set to either future or past timestamps.

We note using `CLOCK_MONOTONIC` which ignores system sleep time and cannot be set by ntpd or user, as discussed in §2.2, would have avoided time bugs in such time arithmetic. However, due to their subtle semantics,

[1] We note this bug scenario can also arise in desktop/server platforms.

many kernel and app programmers make mistakes in using the gettime APIs.

Occurrences. To count the occurrences of this type of time usage, we searched for all the helper APIs that perform time arithmetic at the three software layers. Since time values may also be manipulated with direct integer/-long arithmetic, the number of occurrences counted this way will be an underestimate. Table 1 shows that time arithmetic is extensively performed, with lower bounds of 125, 522, and 236 occurrences in the kernel, framework, and 978 apps, respectively. We note the SITB detection tool we present in §5, however, will capture all Case 3 bugs, whether the time is manipulated using arithmetic APIs or direct arithmetic.

4.2.4 Case 4: Logging

Usage Pattern. In this category, the code obtains the current time and logs it in conjunction with some event, usually for postmortem debugging.

Vulnerability. For such usages, the code between an event and its timestamping forms a time-critical section, as a CPU suspension in between will result in an incorrect timestamp being logged for the event. However, automatically detecting this category of TICSes is challenging, since in general there is no syntactic clue correlating the event and logging. We leave detecting SITBs in this category as future work.

Occurrences. Since it is difficult to count all such usages by searching any APIs, we heuristically assume that if a timestamp call, *i.e.,* gettime(), is not used in one of the above three categories, it belongs to this category. Hence the numbers for Case 4 time usage pattern in Table 1 are overestimates. Table 1 shows timed logging occur 453, 992, and 7209 times in the three software layers.

5 Design

Sleep-induced time bugs occur when a part of a time critical section (TICS) is unprotected. In this section, we explore the design space and present a detection system called KLOCK that automatically finds SITBs in the first three categories of time usages using static analysis.

5.1 Design Space

We explored the use of both software model checking and dataflow analysis, two primary techniques that have been extensively applied to finding software bugs [12, 21, 14, 28, 13, 19]. Both techniques attempt to discover properties that hold for the program or at certain points in the program.

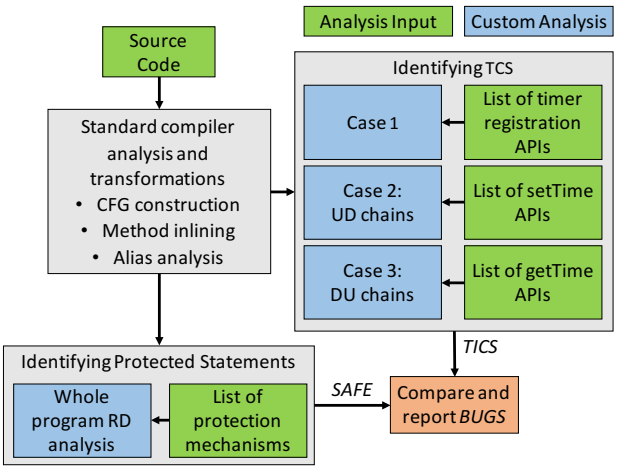


Figure 1: The KLOCK architecture.

Roughly speaking, model checking is well suited to analyses that explore dynamic properties of the system, and data flow analyses are typically used in situations where conservative approximations can be made regarding paths or inter-thread ordering. Which kind of analysis to choose depends on the implementation difficulty and the analysis precision that can be achieved for the property under consideration. As shown in [15], model checking is not always more accurate than data flow analysis and can be more difficult to implement. In the current work, the properties to be modeled (Use/Def information, program paths that may not be covered by a wakelock) corresponded well to what could be accomplished via traditional dataflow analysis. This, combined with the relative ease of implementing these within the LLVM framework led us to use dataflow analysis for our debugging solution.

5.2 Design Overview

Figure 1 gives an overview of the KLOCK design, and the overall detection algorithm. KLOCK takes as input source code and performs two major tasks to detect time bugs. First, it identifies each of the three types of time critical sections of interest by identifying the statements that delimit the TICS and the statements that are contained in the TICS (§5.4). Next, it finds the set *SAFE* of all *safe* statements, *i.e.,* statements that are safe from CPU suspension during their execution (§5.5). The statements that belong to *TICS* and not to *SAFE* are in a TICS that are subject to SOC/CPU suspension while they are executing; they are marked as sleep-induced time bugs and added to the set *BUGS* which are reported.

5.3 Compiler Analyses Used by KLOCK

Our system uses a number of well-known compiler analysis techniques [10]. These include (1) **Points-to anal-**

ysis determines what can be pointed-to by a pointer or reference. Our techniques will use points-to analysis to find the targets of function pointers. (2) **Interprocedural reaching definitions analysis** (RDA) finds all definitions of variable v that can reach a statement that uses variable v. (3) **Use-Def and Def-Use chain construction** links together definitions of variables and uses of those definitions across functions within the program. Use-Def and Def-Use chains are used to follow the flow of data through the program and form the core of our analysis.

5.4 Identifying Time Critical Sections

Identifying time critical sections is, in general, impossible because whether or not a set of statements is a time critical section depends on the intended semantics of the code. We make a key observation that in all three categories of time usage that are vulnerable to time bugs, the start and/or end of time critical sections are marked with a handful of APIs that get/set time or register timer callbacks in the Linux kernel. Therefore, by identifying and informing the compiler of these APIs, *e.g.,* by providing them in a table to the compiler, we can effectively bootstrap the compiler analysis for precisely identifying all such time-critical code sections.

Case 1: Timer Callback. Recall that in this case, the program registers a timer callback for performing a time critical task. The TICS contains the registration, the callback and the critical task performed by the callback.

Algorithm 1 TICS identification for Case 1: Callbacks

Require: Program P, Callgraph C, Timer registration APIs set R
1: Time critical statements $TICS \leftarrow \emptyset$
2: **for all** Statements $S_P \in P$ **do**
3: **if** S_P calls function $F_R \in R$ **then**
4: $TICS = TICS \cup S_P$
5: Callback F_C = getTarget(S_P, C)
6: $TICS = TICS \cup F_C$
7: **end if**
8: **end for**
9: **return** *TICS*

Algorithm 1 gives the pseudo code for detecting callback based *TICS*. To identify the start of a TICS, we identify the small number of callback registration functions R, such as `hrtimer_start` which registers a high-resolution timer in the kernel. Identifying the start of a TICS boils down to matching all calls in the source code against the list of functions in R (line 3).

The next step is to correctly identify the callback function corresponding to every timer registration (line 5). In the kernel, registration is done by passing a pre-defined struct that contains the callback function pointer. For example, in code listing 2, the struct `devpriv->timer` is passed as an argument to `mod_timer` in line 12, and its member `.function` is set to `waveform_ai_interrupt` in line 17. As shown in this example, the callback registration (line 12) and setting function pointer (or defining argument object) can be in disconnected places. Hence, identifying the correct callback function in the kernel code requires pointer analysis.

The end of a TICS should, ideally, be marked by identifying the end of time critical processing inside the timer callback. Since identifying this critical processing can be highly context sensitive, we make a conservative assumption that the TICS ends when the timer callback exits. Hence, we add all the statements in the timer callback to *TICS*, in Algorithm 1.

Case 2: Time Setting. Recall that in this case, the TICS ends with a function call that sets the clock and begins at the point when the time variable used to set the clock was first obtained. Because there are only a small number of APIs that set the clock (*e.g.,* `settimeofday`), a list of them can be maintained in the compiler, and a call to one of these will mark the end of a TICS.

Algorithm 2 TICS identification for Case 2: Set time

Require: Program P, Callgraph C
Require: Statement-variable tuples (S_{ST}, V_t) that set time
1: Time critical statements $TICS \leftarrow \emptyset$
2: **for all** (S_{ST}, V_t) **do**
3: $TICS \leftarrow TICS \cup DEFS^{+}(S_{ST}, V_t)$
4: **end for**
5: **return** *TICS*

Identifying the start of *TICS*, however, is more complicated. This is because the time value that is used to set the clock can be read from the network (*e.g.,* via NTP) or can be directly obtained from the user, *i.e.,* there is no fixed API for obtaining this time value. We do know, however, that the variable obtained from other sources must affect the time variable used to set the clock. Hence, to find the start of Case 2 TICS, we find the transitive closure $DEFS^{+}(S_{ST}, V_t)$ of the use-def chain ($DEFS$ set) where S_{ST} is a statement with an API call to set time and V_t is the variable containing the time for that call. $DEFS(S,V)$ is the set of statements that may have defined V most recently before S. The closure, hence, contains all variable definitions which directly affect the variables containing time at set time API calls. We mark all statements in the closure as part of the *TICS*.

For example in code listing 3, we first mark line 13 (S_{13}), `settime(x)` as the end of a TICS pushing (S_{13}, x) to C_{ST}. $DEFS(S_{13}, x)$ will contain all the definitions of x that reach line 13, *i.e.,* lines 2 and 5 and they are marked as part of the TICS.

Case 3: Time Arithmetic. Recall that timer arith-

metic can be done using either fixed APIs (*e.g.,* `ktime_sub(`t_1,t_2`)`) or general integer arithmetic. This case is challenging because it is not obvious which two time variables are involved in that arithmetic. We make a key observation that only the code between getting the two timestamps used in the arithmetic expression forms a TICS– the arithmetic itself is not a TICS. This observation motivates our detection scheme as follows.

Algorithm 3 TICS identification for Case 3: Time arithmetic

Require: Program P, Callgraph C
Require: Statement-variable tuples (S_{GT}, V_t) that get time
1: Time critical statements $TICS \leftarrow \emptyset$
2: **for all** $(S_{GT}, V_t)(S'_{GT}, V'_t); S_{GT} \neq S'_{GT}$ **do**
3: **if** $USES^+(S_{GT}, V_t) \cap USES^+(S'_{GT}, V'_t) \neq \emptyset$ **then**
4: $TICS = TICS \cup$ all statements between S_{GT}, S'_{GT}
5: **end if**
6: **end for**
7: **return** *TICS*

We first build $USES(S,V)$ which is the set of statements that may use the value of V computed at $S, \forall S \in P$. Now, there are a few APIs to read the current system time, *e.g.,* `getnstimeofday` is used by the kernel to get the time, and `SystemClock.elapsedRealTime` is used by both the Android framework and apps. For each use of these APIs in a statement S_{GT} with the returned time arguments V_t, we find the transitive closure $USES^+(S_{GT}, V_t)$ of the def-use chain ($USES$ set), *i.e.,* we find all statements that are directly or indirectly flow dependent on statement S_{GT}.

If timestamps are obtained at n locations, n closures are computed, corresponding to the n timestamps read. Now, if an arithmetic statement is contained in two different closures, we know that the arithmetic statement contains, and is affected by, variables whose values are either timestamps or a function of the timestamps of the closures it is involved in. All such pairs of timestamps are marked as the start and end of a *TICS* (lines 5-11).

Note that while three or more timestamps can potentially be involved in some arithmetic (we have not seen such cases), the algorithm requires no change as pairwise set intersection will capture all statements in the critical section resulting from such multiple timestamps.

5.5 Identifying Protected Statements

When a protection mechanism is enabled, *e.g.,* a wakelock is held, or interrupts are disabled, all statements until it is disabled are protected by it. Such statements can be detected using a variation of the reaching definitions dataflow analysis as in [25]. The key idea is that enabling and disabling each mechanism can be transformed to assignments of values "1" and "0" to a special mechanism variable, which initially has a value of "0". Afterwards, the state of all protection mechanisms that reach a statement can be easily observed via the reaching definitions dataflow analysis, which determines if the statement is protected. This analysis adds all of the protected statements into the *SAFE* set, which are compared with *TICS* statements to detect SITB as shown in Figure 1.

Table 2: Summary of the analyses in KLOCK.

Analysis	LOC	Time (s)
`clang` Compilation	-	71
`llvm-link` Linking	-	651
Alias analysis	-	164
KLOCK CallGraph	552	2
KLOCK *TICS* Case 1: Callback	164	2
KLOCK *TICS* Case 2: Set time	1489	246
KLOCK *TICS* Case 3: Get time	1101	501
KLOCK *SAFE*: RD Analysis	717	11
KLOCK Other	1196	-
Total	5219	1648

5.6 Limitation and Generality

KLOCK currently does not deal with wakelocks that take timeout parameters; statically finding the end of such protected regions is difficult. KLOCK can detect protected code regions due to the first two mechanisms in §2.1, but not due to suspend callbacks via conditional variable manipulation. Detecting such cases requires involved range analysis (*e.g.,* [27]). Finally, KLOCK does not detect SITBs in timed logging. We leave these as future work.

Although we focus on the Android Linux kernel, the KLOCK design is quite general and can be applied to detect SITBs in the framework, the apps, and other software systems that are vulnerable to system suspension. For example applying the KLOCK design to analyze apps just requires building call graphs that can capture intricate callbacks that cross apps and the framework.

6 Implementation

We implemented KLOCK by adding 5 custom passes on top of the baseline LLVM compiler infrastructure version 3.3 [7], the 4 custom passes discussed in §5.4 cases 1,2, 3, and §5.5, and an additional pass for building call graph of the complete kernel, as shown in Table 2. KLOCK also runs a few standard passes such as alias analysis, control flow graph simplification and a few peephole optimizations. Before running these passes, we manually exposed the relevant APIs for bootstrapping the analyses by annotating the Linux kernel.

LLVM Passes. KLOCK is implemented in C++ in a total of 5.2 KLOC, broken down into implementing different

Table 3: Kernels used in KLOCK evaluation.

Phone	CPU	SoC	Version
Nexus 1	Scorpion	Qualcomm QSD8250	2.6.35.7
Nexus 7	ARM A9	Nvidia Tegra 3 T30L	3.1.10
Nexus 10	ARM A15	Samsung Exynos 5	3.4.5
Nexus S	ARM A8	Samsung Hummingbird	2.6.35.7
x86	x86_64	–	4.1

Table 4: TICS (U)sage, SITB (R)eports generated by KLOCK, and confirmed (B)ugs for each case in the Android kernels. No double-counting of same bug in different kernels.

Category in kernel	Time Callback			Time Setting			Time Aritmetic		
	U	R	B	U	R	B	U	R	B
arch	4	2	1	0	0	0	2	1	1
block	4	4	0	0	0	0	0	0	0
drivers	41	23	3	3	1	0	60	50	48
fs	10	8	0	3	1	0	3	2	0
init	0	0	0	1	1	0	0	0	0
kernel	29	15	0	2	2	0	10	8	2
mm	2	1	0	0	0	0	0	0	0
net	29	29	0	0	0	0	32	9	1
security	0	0	0	0	0	0	2	0	0
sound	4	4	0	0	0	0	6	3	3
tools	1	0	0	0	0	0	4	4	4
Total	123	86	4	9	5	0	120	78	59

passes as shown in Table 2.

7 Evaluation

Our evaluation of KLOCK answers the following questions: (1) How long does it take KLOCK to analyze a large system such as the Linux kernel? (2) Is KLOCK effective in finding sleep-induced time bugs? (3) What causes KLOCK to generate false positive reports? All experiments were conducted on an Ubuntu Linux machine with an Intel 8-Core 2.33 Ghz CPU and 16 GB memory.

Performance. The execution time of KLOCK in analyzing the Linux kernel version 3.1.10 and the time breakdown into all the phases are shown in Table 2. The results show that KLOCK can analyze a large system in a fairly reasonable amount of time.

7.1 Finding Sleep-Induced Time Bugs

Since KLOCK analyzes the entire compiled kernel at once, we apply KLOCK to 5 different kernels, *i.e.,* with different configuration options and/or kernel versions, to increase the coverage of the entire Linux kernel. Table 3 lists the five kernels used in the evaluation, for four popular phones, Nexus 1, Nexus 7, Nexus 10, Nexus S, with default configuration and an x86 kernel with `allyesconfig` that has wakelocks enabled. All four phones have ARM CPUs but have different SOCs.

Table 4 summarizes the bug finding results. Each file containing time related API is counted as one time usage instance, and is counted as an SITB bug if it contains at least one statement in the *BUG* set output by KLOCK.[2] Every number in the table shows the total number of unique instances of usages/bugs across the five kernels. The number of instances and bugs for each of the three time usages is broken down according to the top level directories in the kernel. The usage number for each usage excludes time usages using safe APIs. We observe that time callback and time arithmetic occur 123 and 120 times in the five kernels, but time setting is used rarely, only 9 times by a few kernel components.

After KLOCK generated bug reports, we manually analyzed them and marked them as either false positives or bugs. In total we found 63 bugs out of which we found 14 have been fixed and 7 files with bugs have been removed from later versions of the kernel. We reported the remaining 42 bugs to the Linux kernel mailing lists, and out of the 7 developers who have replied so far, all confirmed the corresponding bugs and accepted our patches. We now describe these 63 time bugs that KLOCK found.

Measuring pulse width. KLOCK found 6 similar bugs in remote control receiver drivers – 4 in the LIRC subsystem `drivers/staging/media/lirc/` and 2 in drivers of streamzap remote `drivers/media/rc/streamzap.c` and DHT11 temperature and humidity sensor `drivers/iio/humidity/dht11.c` which measure the width of received pulse using time arithmetic. In Listing 5, the data being received is encoded in the width of the received pulse (lines 6, 7, 14-18). If the SOC suspends or if the time is reset before line 3, the width of the pulse `deltv` calculated at line 4 will be incorrect resulting into wrong value to be saved in `rx_buf` (line 18).

Measuring clock rate. KLOCK detected 5 bugs in radio, IDE and sound drivers where the drivers calculate the input clock rate using time arithmetic. Due to SITBs, the resulting clock rate measured will be incorrect.

Measuring delay. KLOCK found 4 similar bugs in IrDA chipset drivers, `drivers/net/irda/`, where the driver measures the processing delay `diff` (line 9 in code listing 6) and compares it against minimum turnaround time `mtt`, (line 10). If `mtt` is larger than `diff`, then the frame is transmitted after (`mtt - diff`) microseconds (lines 10-12). If the time is set to a time before computing line 8, then `diff` may become negative, causing unnecessarily large delay in transmitting the frame.

The SASEM USB IR remote control driver `drivers/staging/media/lirc/lirc_sasem.c` ig-

[2] We conservatively use file as a unit in counting bugs to avoid inflating the bug count. In time arithmetic, one start time is often used in arithmetic with many other timestamps, and each could have been counted as a separate bug.

Listing 5: drivers/staging/media/lirc/lirc_sir.c: Using wallclock to measure pulse width.

```
1  static irqreturn_t sir_interrupt(int irq, void *
       dev_id) {
2    if (status & (UTSR0_RFS | UTSR0_RID)) {
3      do_gettimeofday(&curr_tv);
4      deltv = delta(&last_tv, &curr_tv);
5      if (status&UTSR0_RID) {
6        add_read_queue(0, deltv-n*TIME_CONST);//
             space
7        add_read_queue(1, n*TIME_CONST); //pulse
8        n = 0;
9        last_tv = curr_tv;
10     }
11   }
12   return IRQ_RETVAL(IRQ_HANDLED);
13 }
14 static void add_read_queue(int flag, unsigned
       long val) {
15   int newval;
16   newval = val & PULSE_MASK;
17   ..
18   rx_buf[rx_tail] = newval;
19 }
```

Listing 6: drivers/net/irda/nsc-ircc.c: Using wallclock to measure processing delay.

```
1  static netdev_tx_t nsc_ircc_hard_xmit_fir(struct
2    sk_buff *skb, struct net_device *dev) {
3    ..
4    // Start transmit only if there is currently no
         transmit going on
5    if (self->tx_fifo.len == 1) {
6      mtt = irda_get_mtt(skb);
7      if (mtt) {
8        do_gettimeofday(&self->now);
9        diff = (self->now.tv_sec-self->stamp.tv_sec
             )*USEC_PER_SEC + (self->now.tv_usec -
             self->stamp.tv_usec);
10       if (mtt > diff) {
11         mtt -= diff;
12         udelay(mtt);
13       }
14     }
15     // Transmit frame
16     nsc_ircc_dma_xmit(self, iobase);
17   }
18 }
```

nores an input if it arrives in 250 ms since the last input. Because of unexpected discontinuities in wall clock time, the driver may end up ignoring inputs not in the 250 ms range. A similar bug was found in input driver for SoundGraph iMON IR, `drivers/media/rc/imon.c`.

Poll and wait until timeout. Linux control and measurement device interface driver `drivers/staging/comedi/drivers/serial2002.c`, shown in Listing 7, polls serial connected hardware at line 9. If there is no new data, it takes the current timestamp `now` at line 13, calculates `elapsed` at line 14 relative to `start`, obtained at line 4, and breaks the loop if `elapsed` is larger than the `timeout` (lines 15-16). KLOCK correctly flagged time arithmetic at line 14 as a bug since before reading `now` at line 13, if the time is set to a past time stamp, the driver will get stuck spinning in the while loop much longer than the intended `timeout` (typically 1 ms, not shown in Listing 7).

Listing 7: drivers/staging/comedi/drivers/serial2002.c: Using wallclock to poll and wait until timeout.

```
1  static void serial2002_tty_read_poll_wait(struct
       file *f, int timeout) {
2    struct poll_wqueues table;
3    struct timeval start, now;
4    do_gettimeofday(&start);
5    poll_initwait(&table);
6    while (1) {
7      long elapsed;
8      int mask;
9      mask = f->f_op->poll(f, &table.pt);
10     if (mask & (POLLRDNORM | POLLRDBAND | POLLIN
           | POLLHUP | POLLERR)) {
11       break;
12     }
13     do_gettimeofday(&now);
14     elapsed = 1000000 * (now.tv_sec - start.
           tv_sec) + now.tv_usec - start.tv_usec;
15     if (elapsed > timeout)
16       break;
17     set_current_state(TASK_INTERRUPTIBLE);
18     schedule_timeout(((timeout - elapsed)*HZ)
           /10000);
19   }
20   poll_freewait(&table);
21 }
```

The accepted patch for the bug uses `CLOCK_MONOTONIC` which ignores system suspend and cannot be reset by user setting the time or by NTP as discussed in §2.2.

Similar bugs were found in 7 other drivers.

Generating waveform. KLOCK detected both case 1 and case 3 time bugs in the Linux control and measurement device interface driver `drivers/staging/comedi/drivers/comedi_test.c` in code listing 2 which was already discussed in §4.2.1.

Msm, vibrator, and timed gpio drivers. The code snippet in Listing 8, from the msm7k serial device and console driver, is responsible for turning off the UART clock once the transmit buffer is empty. This function first verifies if the clock is on (line 4) and then sets the clock state to `MSM_CLK_REQUEST_OFF` signifying that it is requested to be turned off (line 5). It then registers a timer callback function `msm_serial_clock_off` that must be called after `clock_off_delay` seconds (line 6). This callback function verifies the state of clock to be `MSM_CLK_REQUEST_OFF` (line 12), and checks if the transmit buffer is empty (line 13). If so, the clock is disabled and its state is set to off (lines 14 and 15), otherwise the callback function is rescheduled to get called again in `clk_off_delay` seconds (line 17).

SITB occurs if the CPU suspends before the timer fires and the callback function is executed. In that case, even if the transmit buffer is empty, the UART clock would unnecessarily remain turned on.

The Android kernel exposes a special timer API `android_alarm` which uses high-resolution timer to trigger an event when the CPU is active and additionally also sets an RTC wakeup alarm when the CPU is about to suspend. Switching to Android timer API `android_alarm_init` from `hrtimer_start` (at line 6)

Listing 8: drivers/serial/msm_serial.c: Unprotected use of timer callback wastes energy.

```
1  //request turning off clock once TX is flushed
2  void msm_serial_clock_request_off(struct
       uart_port *port) {
3    clk_off_timer.function = msm_serial_clock_off;
4    if (msm_port->clk_state == MSM_CLK_ON) {
5      msm_port->clk_state = MSM_CLK_REQUEST_OFF;
6      hrtimer_start(clk_off_timer,clk_off_delay,
           HRTIMER_MODE_REL);
7    }
8  }
9  //clock off if TX buffer is empty, else
       reschedule
10 static enum hrtimer_restart msm_serial_clock_off(
       struct hrtimer *timer) {
11   int ret = HRTIMER_NORESTART;
12   if (msm_port->clk_state==MSM_CLK_REQUEST_OFF) {
13     if (uart_circ_empty(xmit)) {
14       clk_disable(msm_port->clk);
15       msm_port->clk_state = MSM_CLK_OFF;
16     } else { //reschedule
17       hrtimer_forward_now(timer, clk_off_delay);
18       ret = HRTIMER_RESTART;
19     }
20   }
21   return HRTIMER_NORESTART;
22 }
```

fixes the SITB since the CPU will be woken up just in time to turn off the UART clock.

Similar bugs were found in vibrator driver `arch/arm/mach-msm/msm_vibrator.c` and timed gpio driver `drivers/staging/android/timed_gpio.c`.

Leaky bucket. The driver for Beeceem WIMAX chipset used by Sprint 4G, `drivers/staging/bcm/LeakyBucket.c`, implements a routine related to the Leaky Bucket algorithm. As shown in the code snippet in Listing 9, function `UpdateTokenCount()` controls the number of packets that can be transmitted in a fixed time period. Line 3 reads the current time in `tv` and line 4 computes the number of seconds passed since the token count was last updated and stores it in `currentTime`. If `currentTime` is non-zero, the current token count is incremented by the number of packets that can be transmitted in `currentTime`, and the last update time is set to current time (line 7).

If the token accounting semantics is to include CPU sleep time, then if the CPU sleeps after line 5, line 6 will be executed after the CPU wakes up and under-calculates the tokens accumulated. If the token accounting semantics is to exclude the CPU sleep time, then the token accounting is correct in the current invocation of the function if the CPU does not sleep before line 3. But if the CPU sleeps after line 3, in the next invocation of function `UpdateTokenCount()`, `currentTime` calculation (line 4) would include the sleep time, again resulting in incorrect token calculation.

Benchmarking and stats reporting. KLOCK detected the sleep bug in the the memcpy benchmark discussed in §3. Similar bugs were detected in 29 other places.

Listing 9: drivers/staging/bcm/LeakyBucket.c: Incorrect token accounting due to SITB

```
1  //Called every time before transmitting packets.
2  static void UpdateTokenCount() {
3    do_gettimeofday(&tv);
4    currTime = tv.tv_sec-pcktInfo.lastUpdate.tv_sec
         ;
5    if(currTime!=0) {
6      pcktInfo.tokens+= pcktInfo.maxRate*currTime;
7      memcpy(pcktInfo.lastUpdate,&tv,sizeof(struct
           timeval));
8      if(pcktInfo.tokens>=pcktInfo.maxBucketSize)
9        pcktInfo.tokens=pcktInfo.maxBucketSize;
10   }
11 }
```

Miscellaneous. 2 other time arithmetic bugs were found in infiniband driver `drivers/infiniband/hw/mlx4/alias_GUID.c`, and storage controller driver `drivers/scsi/3w-9xxx.c`. We skip their details due to page limit.

7.2 False Positives

KLOCK reported 106 time manipulation instances to contain SITBs, which upon manual analysis, turned out to be false positives. We note the false positive rate of 63% is a reasonable tradeoff for the high coverage of static analysis (*e.g.*, [27] reports finding 11 bugs out of 741 reports, [20] reports finding 252 bugs out of 955 reports). We found three reasons that cause KLOCK to generate false error reports:

System suspension does not affect program semantics. We found false positives in cases where the program semantics are not impacted by system suspension during time manipulation. Marvell wireless LAN device driver `drivers/net/wireless/mwifiex/wmm.c`, for example, just calculates a random number by performing time arithmetic. Similarly, the kernel process scheduler registers a timer callback for scheduling a new process at the end of a time slice, but even if the CPU suspends in the middle, the delay in callback execution will have no impact on the scheduler semantics. Reducing such false positives requires understanding program semantics.

System calls. KLOCK flags system calls such as `sys_settime`, `sys_utimes` as bugs. This is because KLOCK only analyzes the entire Linux kernel, and these system calls are effectively wrappers to the actual time setting APIs and are meant to be invoked by user-space programs; by themselves they do not enable any suspension prevention mechanism. Such system call usages can cause sleep-induced time bugs in the user-space programs calling them if they do not use proper suspension prevention mechanisms.

Dependence on system suspension code. Requesting firmware `drivers/base/firmware_class.c`, for example, holds a semaphore shared with the code that disables usermodehelper which lies on the suspension code path.

Reducing such false positives requires tracking the state of all global conditional variables and semaphores shared with all the code on suspension code path. We leave it as future work.

7.3 The Significance of SITBs

SITBs occur in all software layers in the mobile ecosystem. They can impact both performance and program correctness. In particular, out of the 63 bugs KLOCK found, 30 are benchmarking bugs, and the remaining 33 bugs either impact performance (including energy) or correctness of device drivers.

Correctness related. (i) 6 drivers under "Measuring pulse width" decode a received signal by measuring the width of a pulse. SITBs make them measure the width incorrectly, hence reading the received data incorrectly. (ii) 2 drivers under "Measuring delay" incorrectly ignore user input. (iii) 5 drivers under "Measuring clock rate" measure the clock rate incorrectly. These are mostly radio drivers needed to detect the incoming clock rate to decode data. The data decoded will be wrong if the measured clock rate is incorrect.

Performance related. (i) 8 drivers under "Poll and wait until timeout" category cause the driver to spin for a long time, making the device unusable. (ii) 4 drivers under the "Measuring delay" category cause the driver to sleep for a long time, making the device unusable. (iii) 3 drivers, msm, vibrator and timed_gpio, keep the device on longer than necessary, wasting energy.

In summary, none of these bugs crash the kernel, but they are serious bugs affecting the correctness or performance of the kernel.

8 Related Work

Hunting bugs in Linux is a topic almost as old as Linux itself [23]. Sleep-related bugs (in Linux) on smartphones is a relatively new and exciting area. Previous work has focused on sleep bugs that result in energy leaks, or energy bugs. Pathak *et al*. were the first to discuss the significance of energy bugs in smartphones [24] and developed a taxonomy of smartphone energy bugs. In [25], Pathak *et al*. studied no-sleep energy bugs, a class of sleep bugs caused by not releasing wakelocks in apps which causes SOC/CPU to stay awake, and developed a detection tool based on reaching definitions dataflow analysis. In [16], Jindal *et al*. studied sleep conflicts, a class of sleep bugs in device driver code that cause phone devices to stay in an active power state till indefinite due to unexpected SOC/CPU suspension, and proposed a system to perform runtime avoidance of sleep conflict. In [17], Jindal *et al*. developed a taxonomy of sleep disorder bugs, which includes no-sleep, over-sleep and under-sleep bugs. SITBs are first class of over-sleep bugs studied.

Carat [22] treats apps as blackboxes and performs collaborative debugging to identify "energy hog" apps based on observed behavior of an app running on many phones.

In contrast to these previous work, we study a new class of sleep-related bugs, sleep-induced time bugs, that manifest as logical errors and alter the intended program behavior.

Our work relies in part on finding ordering relationships between actions on time-related system calls, variables that are a function of time values, and wakelock acquires and releases. Engler's MC language [11] builds upon the Metal state-machine language and allows writing compiler extensions for static checking of temporal relationships between program actions. As described in §5.4, Algorithms 2 and 3, our techniques require finding transitive closures of use-def chains and, in Algorithm 3, additionally finding the pair-wise intersections of the closures which are beyond statically defined state machines created when compiling MC checks.

We note that [26] and [18], among others, perform static race detection, and in the course of doing this identify the sets of locks held at a location. While it might be possible to take their analysis and adapt it to our needs, we find our simple data flow based algorithm to be sufficient and efficient.

9 Conclusion

This paper presents the first study of a new class of sleep-related bugs on smartphones, sleep-induced time bugs, that can occur in all layers of smartphone software, *i.e.*, the kernel, framework, and apps. A SITB happens when the phone is suspended in the middle of a time critical section that manipulates time and as a result alters the intended program behavior. We characterize the pervasive usage of time usage in smartphone software layers, classify them into four usage patterns, and show their vulnerability to SITBs. We present the design and implementation of KLOCK, a tool that detects SITBs in large systems. KLOCK has aided in finding 63 SITBs in the Linux kernel. We have released KLOCK at `http://github.com/klock-android` for use by smartphone OS developers to test for sleep-induced time bugs.

Acknowledgment. We thank the anonymous reviewers and our shepherd Anthony D. Joseph for their constructive comments which helped to improve this paper. This work was supported in part by NSF grant CCF-1320764 and by Intel.

References

[1] Android time api. http://developer.android.com/reference/android/text/format/Time.html,http://developer.android.com/reference/android/os/SystemClock.html.

[2] Android timer api. http://developer.android.com/reference/java/util/Timer.html.

[3] Decompiling apps. http://siis.cse.psu.edu/ded/.

[4] Kernel manipulating time. http://linux.die.net/man/3/timersub,http://lxr.free-electrons.com/source/include/linux/ktime.h.

[5] Kernel querying time. http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/include/linux/time.h.

[6] Kernel timer api. http://www.ibm.com/developerworks/library/l-timers-list/, https://lwn.net/Articles/429925/.

[7] Llvm compiler infrastructure. https://llvm.org.

[8] [patch] iio: dht11: Use boottime. http://www.spinics.net/lists/linux-iio/msg22706.html.

[9] Speedtest.net. https://play.google.com/store/apps/details?id=org.zwanoo.android.speedtest.

[10] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: principles, techniques, and tools*. Pearson/Addison Wesley, 2007.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. of OSDI*. USENIX Association, 2000.

[12] D. Engler, D.Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of ACM SOSP*, 2001.

[13] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SOSP*, 2003.

[14] Dawson Engler and Madanlal Musuvath. Model-checking large network protocol implementations. In *Proc. of USENIX NSDI*, 2004.

[15] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. of VMCAI*, 2004.

[16] Abhilash Jindal, Abhinav Pathak, Y. Charlie Hu, and Samuel Midkiff. Hypnos: Understanding and Treating Sleep Conflicts in Smartphones. In *Proc. of EuroSys*, 2013.

[17] Abhilash Jindal, Abhinav Pathak, Y Charlie Hu, and Samuel Midkiff. On death, taxes, and sleep disorder bugs in smartphones. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, page 1. ACM, 2013.

[18] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 13–22. ACM, 2009.

[19] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proc. of USENIX OSDI*, 2006.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *Software Engineering, IEEE Transactions on*, 32(3):176–192, 2006.

[21] Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proc. of USENIX OSDI*, 2002.

[22] Adam J. Oliner, Anand Iyer, Eemil Lagerspetz, Sasu Tarkoma, and Ion Stoica. Collaborative energy debugging for mobile devices. In *Proc. of USENIX HotDep*, 2012.

[23] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calves, Gilles Muller, and Julia Lawall. Faults in linux 2.6. *ACM Transactions on Computer Systems (TOCS)*, 32(2):4, 2014.

[24] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging for smartphones: A first look at energy bugs in mobile devices. In *Proc. of Hotnets*, 2011.

[25] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel Midkiff. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proc. of Mobisys*, 2012.

[26] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Locksmith: Practical static race detection for c. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):3, 2011.

[27] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving integer security for systems with KINT. In *Proc. of USENIX OSDI*, 2012.

[28] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proc. of USENIX OSDI*, 2004.

Energy Discounted Computing on Multicore Smartphones

Meng Zhu Kai Shen
University of Rochester

Abstract

Multicore processors are not energy proportional: the first running CPU core that activates shared resources incurs much higher power cost than each additional core does. On the other hand, typical smartphone applications exhibit little parallelism and therefore when one core is activated by an interactive application, computing resources at other cores are available at a deep energy discount. By non-work-conserving scheduling, we exploit energy-discounted co-run opportunities to process best-effort smartphone tasks that involve no direct user interaction (e.g., data compression / encryption for cloud backup, background sensing, and offline bytecode compilation). We show that, for optimal co-run energy discount, the best-effort processing must not elevate the overall system power state (specifically, no reduction of the multicore CPU idle state, no increase of the core frequency, and no impact on the system suspension period). In addition, we use available ARM performance counters to identify co-run resource contention on the multicore processor and throttle best-effort task when it interferes with interactivity. Experimental results on a multicore smartphone show that we can reach up to 63% energy discount in the best-effort task processing with little performance impact on the interactive applications.

1 Introduction

Energy remains the critical resource bottleneck for typical smartphone usage. Due to the slow progress on battery technologies and size restrictions of hand-held devices, battery capacity still limits effective smartphone usage between charges. At the same time, today's popular smartphones are commonly equipped with quad-core or even octa-core processors. Powerful multicore processors put further pressure on the scarce energy resource of a mobile device.

Multicore processors are not energy proportional: the first running CPU incurs much higher power cost than each additional core does. This can be attributed to two reasons. First, modern processors are good at power gating. When the system is completely idle, most parts of the CPU can be shutdown resulting in minimum energy consumption. Second, the sharing of hardware resources on a multicore means that the first running core must activate the bulk of shared resources while additional cores can utilize the already activated resources at much lower cost. This energy disproportionality suggests that a multicore processor is more energy-efficient when more of its cores are utilized at the same time.

Unfortunately, typical smartphone applications are built on event-driven, UI-centric framework and serve only a single user. They do not have sufficient parallelism to utilize multiple CPU cores simultaneously. Recent studies [10, 19] on Android applications show a lack of thread-level parallelism across applications and an over-provisioning of core resources across devices. This implies that smartphone multicore processors often operate at low core utilization resulting in poor energy efficiency. At the same time, when one CPU core is being utilized, computing resources at other cores are available at a deep energy discount.

In this paper, we propose to exploit such energy-discounted co-run opportunities to process best-effort tasks that are useful on a smartphone but do not involve direct user interaction (and thus its time of execution is flexible). One example of best-effort tasks is the file compression and encryption in preparation for backing up the user data to the cloud. Another example is the offline bytecode compilation into native code for optimized application execution in Android. The third example is background sensing and analysis of user's facial expression or eye movements to improve user experience. Work in this paper shows that best-effort tasks may be scheduled to co-run with interactive applications and realize significant energy discount.

The idea of saving smartphone energy by bundling tasks or piggybacking computation on other applications is not new [14, 18]. Unlike previous work, we recognize that optimal energy discount on multicores is only realized when the best-effort task execution does not elevate the overall system power state. Specifically, the best-effort task execution must *not* disrupt the multicore CPU idle state, increase the core frequency, or affect the smartphone's suspension period. In other words, the smartphone's multicore power states should experience no change due to the additional best-effort task execution. We accomplish this objective through careful non-work-conserving CPU scheduling.

While co-execution of applications on multicore processors may improve the energy efficiency, it also risks significant interference on shared hardware resources, memory bandwidth and last-level-cache space in particular, and thereby leads to poor interactive application performance and degraded user experience. To mitigate such contention, we use available processor performance counters to monitor memory bandwidth usage during the co-execution, and throttle the best-effort task when it interferes with the foreground application interactivity.

The rest of this paper is organized as follows. Section 2 elaborates on multicore energy disproportionality and available smartphone best-effort tasks that motivate our work. Section 3 presents our design of energy-discounted computing and resource contention mitigation on multicore smartphones. Section 4 describes our implementation on the Android platform. Section 5 evaluates the energy saving of best-effort task executions and the impact of user interactivity between alternative approaches. We also perform trace-based application analysis to demonstrate the abundance of energy-discounted computing opportunities in various smartphone usage scenarios. We present related work in Section 6 before concluding the paper in Section 7.

2 Motivation

2.1 Multicore Energy Disproportionality

CPUs have traditionally been the biggest energy consumer in the computer system and are not energy proportional. Thanks to many innovations, they are now much improved. Today, multicore processors have very sophisticated power states which can be dynamically adjusted to adapt to different workloads. Specifically, dynamic voltage and frequency scaling (DVFS) is used to achieve a wide range of performance / power settings when the system is active. During the idle period, clock gating and power gating are heavily utilized to power down various parts of the processor in order to achieve low power consumption. These techniques enable the CPU to scale their power consumptions relatively well in relation to their utilizations, making them probably the most energy proportional hardware component in the current computer system.

However, making good energy proportional hardware remains difficult and current CPU's energy proportionality is far from perfect. This is particularly true for multicore processors. Figure 1 shows power consumptions of several multicore smartphone / tablet platforms when different number of cores are active. On all platforms we can observe a disproportionate power jump when activating the first core of each multicore processor. Specifically, we can see that activating each additional cores typically consumes less than half of the power comparing to that of the first core. This is substantial given the small profile of the mobile device.

This energy disproportionality is mostly due to the aggressive hardware sharing. In order to drive down cost, reduce footprint and save power, modern multicore processors share substantial hardware components between cores. CPUs on one socket usually share the oscillator and power rail which forces each CPU to operate at the same frequency. As a result, multicore processors can achieve high energy efficiency during heavy parallel processing. However, if the workload can not scale to take advantage of available cores, the entire socket will have to be kept at certain frequency and voltage level to accommodate a few cores' performance needs resulting in a waste of energy.

Besides limiting the capability of active performance / power scaling, hardware sharing also affects the processor idle state. Table 1 lists the available CPU idle states on the Huawei Mate 7 smartphone. As you can see, individual core idle state (C1) does not have much impact on the overall power consumption. Maximum power saving is only achieved through continuous and simultaneous CPU sleeps (C2) [29]. Again, hardware sharing plays an important role here. For example, L2 cache and related memory subsystem can only be shutdown when the whole CPU cluster is completely idle.

We are aware that various heterogeneous architectures have been proposed to improve the CPU energy proportionality. For example, chips with asymmetric clocking capabilities are able to set different frequencies for different cores, realizing more flexible performance / power scaling. Also, chips equipped with CPUs of different micro-architectures (e.g., ARM big-LITTLE) are used to mitigate the performance vs. power dilemma. However, these techniques do not completely eliminate the energy disproportionality. Hardware sharing is and will continue to be one of the fundamental design principles of multicore processors. Consequently, CPU energy disproportionality will remain a reality that computer systems have to live with in the foreseeable future.

To summarize, modern multicore processors can achieve high energy efficiency when doing heavy parallel processing or in complete idle states. But due to the aggressive hardware sharing, they are very inefficient when dealing with workloads of limited parallelism. Unfortunately, it is well known that typical smartphone applications lack the parallelism to utilize the increasing number of cores available to them. This creates opportunities for the mobile system to make use of the extra computation resources to complete certain tasks at an energy discount.

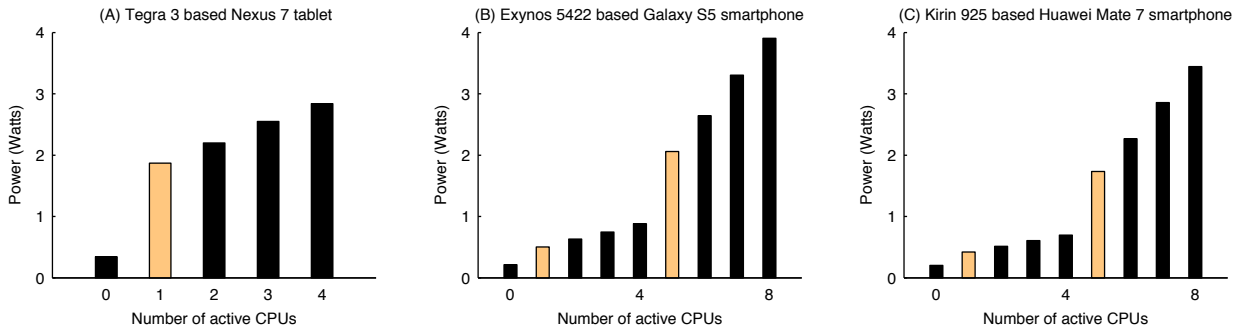


Figure 1: Disproportionate multicore power consumption on the number of active CPUs. The Tegra 3 chipset on Nexus 7 tablet (A) contains a quad-core processor. Both Exynos 5422 (B) and Kirin 925 (C) chipsets contain two heterogeneous quad-core processors. We remove the device battery and use Monsoon Power Meter [4] to measure the whole system power consumption. Devices are put into early-suspend mode where the display and the touchscreen are turned off.

State	Name	Power	Target residency	Description
C0	Wait for interrupt (WFI)	403 mW	1 nSec	Processor is clock gated but can respond to cache / TLB maintenance (e.g., L2 snoop) requests without exiting the WFI state.
C1	Individual powerdown	365 mW	1 mSec	Processor is power gated. All state including L1 cache content is lost and the processor is removed from the coherency protocol.
C2	Cluster powerdown	214 mW	4 mSecs	Can only be entered when all processors are in individual powerdown mode. All state including the L2 cache content is lost.

Table 1: CPU idle states available on the Huawei Mate 7 smartphone. A state's target residency is a value defined in the corresponding *cpuidle* driver in the Linux kernel. It indicates the minimum time period during which the CPU expects to remain idle so that it is worthwhile to enter the state.

2.2 Best-Effort Tasks

We define best-effort tasks as application workloads that are meaningful to the user but do not involve direct interaction and thus have loose quality-of-service requirements. As mobile phones are increasingly used for a variety of purposes, best-effort tasks are becoming common in day-to-day uses. Here are a few examples.

Upload and download operations are common on smartphones. Syncing data with cloud storage services, posting on social websites and software installation / update are typical smartphone usages. Some of them can be delayed to a certain extent. Although significant energy consumption comes from the transmission module, CPUs also consume substantial energy during the process. For example, data compression / decompression requires heavy computation. And encryption / decryption are almost mandatory nowadays which also involve nontrivial CPU processing.

System maintenance work sometimes can also be treated as best-effort tasks. For instance, Android / Linux uses *kswapd* daemon to scan for memory pages that can be swapped out to free up space. Another example is a system daemon called *dhd_dpc* which analyzes network packets and scans for Wi-Fi hotspots. In addition, during application installations, Android would optimize the downloaded packages by recompiling the bytecode for better native performance. All these require substantial CPU processing. Some of them may have timing constraint (e.g., memory management). However, their completions often only matter when the user is also actively using the phone, in which case discounted computation opportunities are likely to be abundant (Section 5.4).

Background sensing is also a suitable best-effort task. Previous work [14] has shown that delaying sensing activities to overlap with other application executions can be more energy efficient. With our technique, the bundled execution can reap even more energy discount. Re-

cent trends also suggest more creative ways of sensing. For example, using camera sensors to analyze user's facial expressions or eye movements [1] may improve user experience. Due to privacy issues, it is beneficial to perform these analysis locally which will put pressure on the device battery life. Since these sensing activities often overlap with user interactive tasks, our technique can be used to substantially lower the energy cost.

Proactive tasks are done predictively to improve user experience. As smartphones getting "smarter", these tasks are becoming increasingly common. For example, Siri can provide recommendations, news and applications "even before you ask" [3]. Previous work [25] also suggests to pre-launch applications to hide user perceived delay. These tasks often do not have hard deadlines and thus can benefit from our technique to save energy.

3 Energy Discounted Computing

Given the energy disproportionality of smartphone multicore processors and the lack of parallelism in typical mobile applications, it is possible to get a deep energy discount by co-scheduling best-effort tasks with the interactive application. However, achieving maximum energy discount without impacting the user experience requires careful system control.

3.1 Power State Preservation

During the execution of interactive applications, the CPU will dynamically adjust its power states to meet the application performance needs. The key principle of reaching the optimal energy efficiency is to utilize the additional (otherwise idle) processor resources without elevating the overall CPU power state.

- *CPU idle state, or ACPI "C" state [24]:* On smartphones, there are often long idle gaps between user interactions during which the user is consuming the content on the screen while all CPUs enter deep sleep state. As simultaneous and continuous sleeps can save a lot of energy [29], it is crucial to keep best-effort tasks from disrupting these idle periods. On the other hand, during active application executions, due to lack of parallelism, idle CPUs will often enter per-core idle states. These shallow sleep states, as we mentioned in Section 2.1, do not save much energy. Thus these idle cores can be utilized to run best-effort tasks at an energy discount. To achieve this, the CPU scheduler needs to schedule best-effort tasks opportunistically in accordance with interactive applications and therefore non-work-conserving CPU scheduling may be necessary. Specifically, the system should schedule best-effort tasks on idle cores only if there is at least one sibling CPU being actively utilized by the interactive application. Otherwise it should enter idle state even when best-effort tasks are ready to run.

- *Core frequency state, or ACPI "P" state:* Modern CPUs use DVFS to quickly adjust power levels to conserve energy and meet performance needs of different workloads. In our co-run scheme, the system should avoid raising the CPU frequency / voltage levels for best-effort tasks. Otherwise, the extra energy consumption will negate the energy discount and the system may well consume more energy than running each task individually combined. At the same time, such caution should not affect the performance of interactive applications. In other words, the CPU frequency adjustment should only focus on the needs of interactive applications and ignore the presence of best-effort tasks.

- *Smartphone suspension state, or ACPI "S" state:* Systems in the suspension state consume very little energy by shutting down most parts of the hardware, including the CPU and memory. On some platforms (notably Android), applications can prevent system suspension by making explicit requests to the operating system. It is important that, in our design, best-effort tasks are not permitted to make such requests. The system should be able to enter the suspension state regardless of best-effort tasks.

To summarize, realizing the maximum energy discount requires judicious control of various aspects of the system to prevent best-effort tasks from elevating the system-wide CPU power states. In other words, best-effort tasks should be *invisible* to the system when making CPU power state adjustments.

3.2 Resource Contention Mitigation

Carefully running best-effort tasks along with interactive applications can bring significant energy savings. However, such savings should not sacrifice user experience. In particular, performance of interactive applications should not be affected.

Co-running tasks on a multicore can potentially slow down each other due to resource contention. This is further exacerbated in our system due to its scheduling strategy—best-effort tasks are intentionally scheduled to run hand in hand with interactive applications.

One easy mitigation is to adjust the CPU scheduling priority. Various parameters are available for this purpose (e.g., nice values and CPU shares on Linux). In our

design, due to the clear importance of interactive applications, we choose to grant absolute priority to them—they are always picked by the scheduler before best-effort tasks. In other words, within one CPU, best-effort tasks can only be scheduled when there is no interactive task waiting.

Absolute priority can eliminate contention on CPU time and mitigate private cache and TLB pollution. However, due to the hardware resource sharing on multi-core processors, contention could also result from shared hardware resources like last-level-cache space and memory bandwidth between cores. Our system uses a simple contention identification approach. Specifically, we monitor the last-level-cache miss rate using the available performance counters. Contention is identified if the miss rate reaches a threshold that suggests memory bandwidth saturation. We acknowledge a limitation of our approach—last-level-cache space contention that does not lead to memory bandwidth saturation will not be identified. Comprehensively identifying cache space contention would be challenging and it generally cannot be accomplished by online monitoring of performance counters alone.

Once the contention is identified, the common approach is to throttle the antagonist (low priority tasks in the contention) executions. This can be done most efficiently on platforms that support certain hardware features such as CPU duty-cycle modulation or asymmetric frequency clocking. Unfortunately, these hardware features are not widely available on today's smartphone processors.

Our system relies on the CPU scheduler to throttle the best-effort tasks. Comparing to the above techniques, however, this is more coarse grained. We can only assert control in the granularity of a CPU quantum (otherwise risk extra scheduling overhead). Fortunately, closely monitoring the contention through performance counters could help time the throttling control more accurate, making this approach quite effective in practice.

4 Implementation

We have implemented our co-run scheme on Huawei Mate 7 smartphone running Android 4.4 and Linux 3.10.30. Our entire modification resides in the Linux kernel.

We use Linux control groups (cgroup) to identify best-effort tasks in the kernel. During system boot, a CPU control group named *best-effort* is created under the cgroup root hierarchy. Best-effort tasks can then be easily added to this group by interacting with the cgroup virtual file system.

Non-work-conserving CPU scheduling We modify the Linux complete fair scheduler to maximize CPU idle state energy saving. Our scheduling policy requires coordination between sibling CPUs. To avoid expensive cross-CPU interrupts and synchronizations, we implemented these communications asynchronously. Specifically, a CPU that wants to schedule best-effort tasks is responsible for checking its siblings' state. We have each CPU maintain a flag indicating its current scheduling state with the following four values:

- *BUSY* indicates the CPU is running normal tasks (e.g., interactive applications),
- *IDLE* indicates the CPU is in idle state (regardless of the level of idle state),
- *BEST-EFFORT* indicates the CPU is running best-effort tasks,
- and *UNDEF* is a transient state (e.g., during context switches).

These per-core flags are cache line aligned to avoid possible false sharing. Although the asynchronous flag read may return stale value under data races, it is likely to be corrected at the next scheduling opportunity. When the scheduler is picking next task to run, normal tasks have absolute priority and are always picked before best-effort ones. If there are only best-effort tasks left in the run queue, the scheduler will first check its siblings' state. If any of them is currently *BUSY*, it will proceed to schedule one of the best-effort tasks. Otherwise, it will enter idle state directly.

Frequency preservation Kernel *cpufreq* governor is responsible for adjusting the CPU frequency. It can come in different flavors but the process usually involves tracking the system load and making adjustments according to some fine-tuned parameters. The load is calculated on a per-core basis by looking at the CPU busy time during the past epoch. Governors typically raise the frequency according to the need of the most heavily loaded CPU (if per-core frequency setting is not available).

To prevent best-effort tasks from affecting the CPU frequency, we track their CPU usage and subtract that from the total CPU busy time when calculating the load. This, along with the absolute priority modification in the scheduler, essentially make best-effort tasks invisible to the CPU governor. While governors completely ignore best-effort tasks, they can still respond to the need of interactive applications just like before.

These modifications reside in the generic governor framework thus individual governor change is not needed.

Suspension management On Android, *wakelock* is used to govern the system suspension state. Applications that want to keep the system awake need to make explicit request to the kernel and grab a wakelock. According to our co-run policy, best-effort tasks should not hold any wakelocks. We modify the wakelock kernel *sysfs* interface to reject any requests made from best-effort tasks.

Contention-triggered throttling Our performance counter based throttling strategy is implemented as a loadable kernel module. We assess the memory bandwidth usage by monitoring the L2 (last-level) cache miss rate. Specifically, we select two events in ARMv7 performance monitoring unit: *ARMV7_A15_PERFCTR_L2_CACHE_REFILL_READ* as L2 cache read miss and *ARMV7_A15_PERFCTR_L2_CACHE_REFILL_WRITE* as L2 cache write miss. Combined they approximate the total access to the main memory. The module triggers periodic interrupt every 20 ms to collect and update the counter statistics. We read the counter value directly from the registers and take care of the overflows. Before picking best-effort task, the CPU scheduler is required to check the latest L2 cache miss rate. If the rate is above certain threshold, the best-effort task will not be scheduled. Similar to our other scheduler modifications, the counter maintenance and lookups are performed in an asynchronous way to avoid the overhead of cross-CPU interrupts and synchronizations.

Our modifications (including the periodic performance counter reading) incurs less than 1% performance overhead for all our benchmarks described in Section 5.1.

5 Evaluation

In this section, we evaluate our techniques on a real device with realistic benchmarks. Section 5.1 introduces our evaluation setup. Section 5.2 evaluates the system energy efficiency. Section 5.3 assesses the effectiveness of our contention mitigation measures. Section 5.4 provides a trace-based application study to demonstrate the abundance of energy-discounted computing opportunities in various smartphone usage scenarios.

5.1 Evaluation Setup

Experimental device We use Huawei Mate 7 smartphone. It was released in October 2014 and is equipped with a Hisilicon Kirin 925 SoC which contains an ARM big.LITTLE octa-core CPU. We use the big cluster in our evaluation. It has four 1.8 GHz ARM Cortex-A15 cores. Each core has its own 32 KB/32 KB L1 instruction and data cache and all cores share a 2 MB L2 cache. It has 2 GB LPDDR3 memory with a bandwidth of 12.8 GB/s.

Power measurement In order to do precise power measurement, we remove the smartphone's battery and connect its power pins to the Monsoon Power Meter [4] which acts as an external power source and measures the phone's overall power consumption. The power meter is able to sample the current at 5 kHz. We turn off hardware components like GPS, cellular and dim the display to the minimum brightness. WiFi is kept on for the purpose of controlling the phone through the host machine without the USB connection (which will disturb the power measurement).

Interactive applications We assemble a suite of benchmarks to represent typical interactive and best-effort application co-run scenarios. Two representative interactive applications are chosen. Bbench [12], a widely used web browsing benchmark which automatically loads and renders locally cached popular websites. It measures the browser performance by tracking the JavaScript *onLoad* event which is triggered once a webpage is fully rendered. We run it using the Android default web browser. Another interactive application is Angry Bird, a popular mobile casual game.

For the co-run experiments, it is important that we are able to measure the *interactivity* of the interactive application. For applications like web browser, the interactivity can be defined as time needed to complete certain tasks. For Bbench we use the aggregated webpage rendering time to measure its interactivity. On the other hand, for games like the Angry Bird, the interactivity is only defined by how responsive the application is. Frames-per-second (FPS) is a more relevant metric. We use GameBench [2] to measure its FPS.

Best-effort applications We select five applications as best-effort tasks.

Spin, a CPU intensive microbenchmark that calculates the n-th triangular number by summation. We choose this microbenchmark to illustrate the optimal co-run scenario—a CPU intensive workload with little memory activity.

Compression compresses a set of files using bzip and *Encryption* encrypts them using the AES encryption. These two are chosen to mimic user download and upload activities.

AppOpt optimizes Android application packages by recompiling them into native code. This is chosen to represent typical deferrable system work.

FaceAnalysis is an in-house developed application that analyzes input faces. It uses Stasm [17], an active shape

model based library, to process images and extract positions of landmark features. This is particularly useful in facial expression analysis. We use it to represent emerging passive sensing applications. To make the experiment reproducible, we use locally cached face images as its input.

Input Workload Application workloads are carefully chosen such that the executions of the interactive application and the best-effort task can mostly overlap with each other when using our co-run strategy. Specifically, Bbench are configured to load 15 websites with two seconds delay (to mimic user think time) between each website. The whole session takes roughly 44 seconds to complete. Angry Bird, on the other hand, is played for 42 seconds. Best-effort tasks are launched in the background shortly after the interactive application starts and the amount of the work is configured such that they can finish right before the interactive application ends under the most strict (throttling-based) best-effort task scheduling policy. To make experiments reproducible, we use *RERUN* [11], a record and replay tool for the Android operating system, to automate the test flow. User interaction sessions are recorded into a sequence of touch and system events. Later, these events are sent back to the phone to replay user interactions with precise timing and accuracy.

5.2 Energy Efficiency

To evaluate the energy efficiency of our system when running best-effort tasks with interactive applications, we run Bbench and Angry Bird with each of the five best-effort workloads. We run each pair under two different scheduling strategies:

- default, where there is no change to the original system behavior;
- power-states-preservation scheduling, where our non-work-conserving scheduling techniques are used.

Figure 2 and Figure 3 show the result. Energy discount (σ) of the best-effort task is calculated as

$$\sigma = \frac{E_{\text{best-effort}} - (E_{\text{co-run}} - E_{\text{interactive}})}{E_{\text{best-effort}}} \qquad (1)$$

where $E_{\text{best-effort}}$ is the amount of energy consumed by the best-effort task running alone under the default system setting, $E_{\text{co-run}}$ is the total system energy consumption of the co-run execution and $E_{\text{interactive}}$ is the total system energy consumption when running the interactive application alone. Each of our energy metrics measures the active energy—those consumed above the system idle power consumption.

The result clearly shows that our system can realize deep energy discount in all co-run scenarios, ranging from 23% to 71%. We attribute this to the fact that the overall CPU power states are preserved—the execution of the best-effort task is completely hidden behind the interactive application power profile.

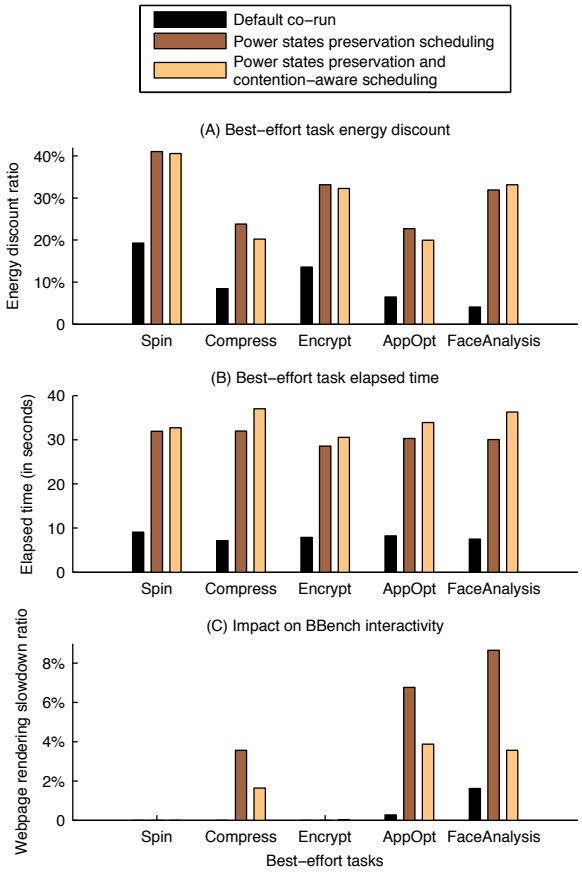


Figure 2: Experimental results of running interactive application BBench web browsing with various best-effort tasks under different scheduling strategies. We show the best-effort task energy discount (A), best-effort task elapsed time (B), and impact on BBench's interactivity (webpage rendering slowdown) (C).

This is further illustrated in Figure 4. When Bbench running alone, the current trace shows the typical burst-then-idle pattern that is common on smartphones due to the long user think time between interactions. During these idle periods, the system is able to enter deep sleep states to conserve energy (trough in the current waveform). However, best-effort tasks, without any control, will disrupt these deep sleep states. In addition, during the burst period, simultaneous executions of both tasks would increase the system load and drive up the CPU fre-

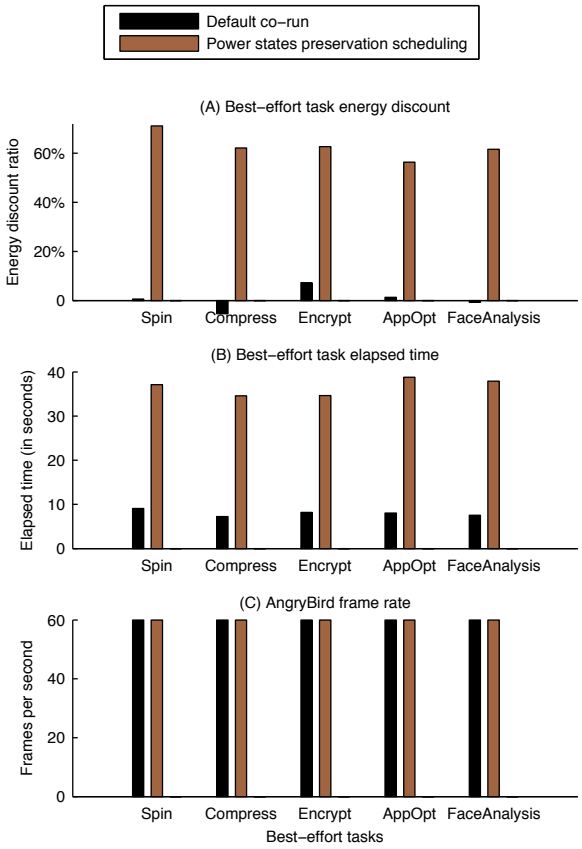


Figure 3: Experimental results of running interactive application Angry Bird with various best-effort tasks under different scheduling strategies. We show the best-effort task energy discount (A), best-effort task elapsed time (B), and Angry Bird's frame rate (C).

quency. In both cases, the system overall power states are elevated, resulting in more energy consumption. These are shown in the current waveform of the default co-run execution. With our power-state-preservation scheduling, both kinds of disruption can be avoided. The best-effort task is piggybacked by the interactive application execution during the burst period, resulting in improved energy efficiency.

It is worth noting that, in Figure 2, the default co-run strategy can also provide some energy discount. This is mostly due to the fact that we intentionally overlap the two application executions by launching them roughly at the same time. Thus a portion of the best-effort task execution happens to be able to utilize some computation resources without elevating the overall CPU power state. In other words, the resulted energy discount is undependable at best. It completely depends on the application characteristics and how the user interacts with them. In fact, in some of our tests, the default co-run strategy can result in more energy consumption than running each task individually combined. Given the typical burst-then-long-idle smartphone usage pattern, the default co-run strategy is likely to perform poorly in most practical scenarios. Our system, on the other hand, always preserves the CPU power states and thus is able to *consistently* provide high energy discount under all circumstances.

Our non-work-conserving scheduling strategy inevitably reduces the system resource utilization and leads to longer execution time of best-effort tasks. Fortunately, best-effort tasks do not involve direct user interaction thus their time of execution is somewhat flexible. The saved energy, on the other hand, could extend the smartphone battery life and let user use their phones more freely which could greatly improve the user experience.

5.3 Throttling-based contention mitigation

As shown in Figure 2, there are non-negligible slowdown on the interactive application when doing power-states-preservation co-run scheduling. In this part of the evaluation, we assess the effectiveness of our throttling-based contention mitigation technique.

We first focus on two application pairs which experience large interactivity slowdown: Bbench+AppOpt and Bbench+FaceAnalysis with 6.77% and 8.66% slowdown respectively. Different L2 miss rate throttling thresholds are used to evaluate their impact on the system performance. Table 2 shows the result. The throttling technique, with properly set threshold, proves to be effective in resource mitigation and minimizing the interactivity slowdown. With L2 miss rate threshold set at 15 misses/μSecs, both the best-effort task elapsed time and the interactivity slowdown remain similar to the non-throttling based scheduling strategy. This means that the throttling mechanism is probably not triggered and a lower threshold is needed. With lower L2 miss rate thresholds, the best-effort task begins to see increased elapsed time while the interactive application performance is improved. This suggests that the system is throttling best-effort tasks while the memory bandwidth is under pressure as it reaches the L2 miss rate threshold. This in turn helps to improve the interactive application performance by reducing the resource contention caused by best-effort task.

However, the benefit is diminished above 10 misses/μSecs. Beyond that, there is very little improvement and even negative impact on the interactive application and the elapsed time of the best-effort task increases dramatically. For a threshold of 6 misses/μSecs, the FaceAnalysis benchmark could not even finish within the 44 seconds Bbench session. This implies that 10 misses/μSecs L2 miss rate is a good indicator of the memory bandwidth saturation and any

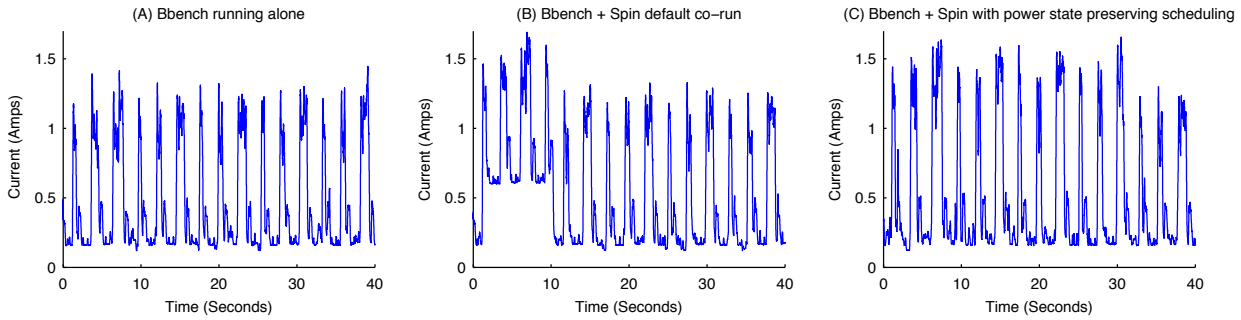


Figure 4: Current trace of Bbench running alone and Bbench+Spin co-run under different scheduling strategies. In the default co-run test (B), the spin task takes 9.07 seconds to finish. Under the power states preserving scheduling strategy (C), the same task takes 31.91 seconds.

lower thresholds can only bring unnecessary slowdown to the best-effort task without benefiting the interactive application.

Next, we apply the contention-aware scheduling technique to all Bbench co-run pairs. The throttling threshold is set at 10 misses/μSecs. Figure 2 shows the result. Besides reducing the Bbench slowdown, we can also see that the throttling technique does not affect the energy savings. In addition, for Spin and Encryption where there is no slowdown of the interactive application, we observe little changes on the best-effort task elapsed time. This suggests that the scheduler is truly contention-aware—it only activates the throttling when there is resource contention in the system.

5.4 Trace-based Application Analysis

In this section we conduct a trace-based application analysis to study the amount of energy discounted computing opportunities in typical smartphone usage scenarios.

Eight popular applications are selected, their detailed descriptions are listed in Table 3. For fair comparisons, each usage scenario lasts for one minute.

We use Linux debugfs-based kernel trace facility to collect CPU frequency and idle state transition events and then calculate the amount of energy discounted computing cycles based on our power-states-preservation scheduling policy. Specifically, if at least one CPU is actively running at certain frequency, other idle CPUs are counted to be able to provide energy discounted computing cycles at that frequency.

For each usage scenario, we measure the abundance of energy discounted computing opportunities in relation to the active CPU usage of the corresponding interactive application. Further, we convert the amount of discounted cycles into meaningful best-effort task workloads by normalizing it to the amount of CPU cycles needed to finish a unit of work (e.g., analyzing one frame of face or encrypt one minute standard resolution video).

(A) Bbench + FaceAnalysis

Throttling threshold (misses/μSecs)	Best-effort task elapsed time	Bbench slowdown ratio
7.5	42.65 Secs	3.94 %
10.0	36.25 Secs	3.57 %
12.5	32.42 Secs	6.58 %
15.0	30.20 Secs	8.73 %

(B) Bbench + AppOpt

Throttling threshold (misses/μSecs)	Best-effort task elapsed time	BBench slowdown ratio
6.0	43.10 Secs	4.20 %
7.5	36.85 Secs	3.72 %
10.0	33.87 Secs	3.88 %
12.5	29.62 Secs	5.80 %
15.0	29.77 Secs	6.81 %

Table 2: The impact of L2 cache miss rate throttling threshold when doing power states preservation and contention-aware scheduling. We show differences in best-effort task elapsed time and Bbench slowdown ratio for two scenarios: (A) Bbench co-running with FaceAnalysis and (B) Bbench co-running with AppOpt.

Table 4 shows the result. As you can see, there are substantial energy discounted computing opportunities in all application categories. This is consistent with the earlier observation that typical smartphone applications lack the parallelism to utilize multicore CPUs. This study demonstrates the potential of exploiting the opportunities enabled by the lack of parallelism in smartphone applications to process useful best-effort tasks at a deep energy

Category	Description
Web Browsing	In Chrome, go to Yahoo.com and browse three top news.
Video Streaming	In YouTube, watch a short HD video for one minute.
Gaming	Play casual game Subway Surf for one minute.
Navigation	In Google Maps, search several nearby attractions and get their directions.
Messaging	In Hangout, open two conversations, type and send two messages.
Social Network	In Facebook, load personal timeline, refresh for friend feeds and browse three posts.
Camera	Use the native camera app to take a minute long video.
Music Streaming	In Google Play Music, stream a song for one minute.

Table 3: Description of eight application scenarios used in the trace-based application study.

Category	Abundance of discounted CPU cycles (multicore)	Abundance of discounted CPU cycles (single-core)	Equivalent work of FaceAnalysis (frames of faces can be analyzed)	Equivalent work of Encryption (minutes of video can be encrypted)
Web Browsing	1.63	0.66	30	21
Video Streaming	2.41	0.85	4	3
Gaming	1.61	0.65	21	15
Navigation	2.42	0.85	13	9
Messaging	2.88	0.97	3	2
Social Network	1.88	0.72	12	9
Camera	2.10	0.77	5	4
Music Streaming	1.63	0.66	7	5

Table 4: Results for the trace-based application study. Each usage scenario lasts for one minute. Abundance of discounted CPU cycles is the ratio of energy discounted CPU cycles to the active CPU cycles used by the corresponding interactive application. In the second column (marked with multicore), energy discounted cycles on all CPUs are counted, assuming the best-effort task has perfect parallelism to utilize all idle CPUs. In the third column (marked with single-core), energy discounted CPU cycles are only counted on one of the eligible CPUs, assuming the best-effort task has no parallelism. We use single-core cycles to calculate the equivalent work of best-effort tasks in column four and five.

discount.

6 Related Work

Smartphone power characterization and energy management has received a great deal of research interests. Using an extensive smartphone power instrumentation platform, Carroll and Heiser [5] developed a power model of various smartphone components and identified promising directions to improve power management. They [6] further suggest that, on a multicore smartphone, CPU cores should be kept online as long as there is work for them. AppScope [26] monitors kernel as well as application activities and correlates them with the smartphone power usage. Song et al. [22] optimized smartphone energy efficiency by lowering CPU frequency when user facing tasks are completed (e.g., display updates finish). Martins et al. [16] monitor and intercept smartphone background activities while the system is in suspension state to extend the battery life. In this paper, we present a new approach to improve smartphone energy efficiency by carefully running best-effort tasks together with interactive applications on a multicore to realize deep energy discounts.

Previous research has recognized the efficiency benefit of piggybacking or co-running background work while the system is active with primary tasks. A classic example [15] is to perform disk work "for free" if such work happens to lie in the disk head rotation and seek path to serve the foreground requests. In the context of mobile systems, Lane et al. [14] showed that performing sensing work while a smartphone is otherwise already active saves substantial wakeup power costs. Nikzad et al. [18] developed an annotation language that demarcates power-hungry executions for delayed execution when the device enters an active state. In this paper,

we make new contributions on energy-efficient multicore piggyback execution by non-work-conserving scheduling that preserves the system power states as if the best-effort task does not run.

Work-conserving schedulers that always utilize available resources when there is work to do is generally desirable for high resource utilization. However, previous research has found the benefits of non-work-conserving scheduling in particular contexts. In disk scheduling, the anticipatory scheduler [13] may keep the disk idling for a short period of time even when there are pending operations. It does so in anticipation of a new I/O operation from the process that issued the just completed operation, which often requires little or no seeking from the current disk head location. In the context of hardware multithreading processors, Fedorova et al. [7] found that running fewer threads than the number of processors may reduce resource contention and improve performance. In this paper, we use non-work-conserving scheduling to run best-effort tasks only when it does not elevate the power states of a smartphone.

There exists a large body of prior work on characterizing smartphone workload behaviors. Gao et al. [10] analyzed a broad range of mobile applications and found that they exhibit little thread-level parallelism and thereby are unable to effectively utilize multiple CPU cores. Seo et al. [19] showed that the lack of thread-level parallelism also prevents mobile applications from effectively utilizing the heterogeneous (big and little) multicore processors. Shingari et al. [21] have identified that co-running multiple mobile applications may yield substantial contention on shared multicore resources. These identified characteristics of smartphone workloads motivate our work of improving execution parallelism and mitigating potentially resulted performance interference.

Multicore performance and power management has been an active area of work for general computer systems. In particular, many techniques were proposed to manage and isolate the shared multicore resources, by partitioning the shared on-chip cache [23, 28], contention-easing scheduling [9, 27], and execution timeslice adjustment [8]. Multicore power disproportionality was also recognized in server power modeling [20]. Mobile systems present new circumstances for multicore performance and power management. First, a lack of thread-level parallelism results in poor multicore energy efficiency on smartphones. Second, the co-existence of interactive and best-effort applications presents differential quality-of-service requirements that complicate resource management.

7 Conclusion

This paper demonstrates the feasibility and benefits in running best-effort tasks on multicore smartphones for an energy discount. The work is motivated by the energy disproportionality of multicore processors and the lack of parallelism in typical smartphone applications. Due to hardware resource sharing, the first running CPU on multicore processors could incur high power cost while each additional core can be used at a much lower energy cost. On the other hand, smartphone applications exhibit little parallelism, leaving room for energy discounted computing on the additional cores.

We propose to exploit these opportunities by running best-effort tasks—tasks that are useful to the user but do not involve direct user interactions. Our contribution lies in the recognition that maximum energy discount can only be realized when overall system power states are preserved. Specifically, the multicore CPU idle state, the processor frequency and the system suspension time should not be affected by the presence of best-effort tasks. We apply careful non-work-conserving CPU scheduling to achieve this goal. In addition, to deal with the interactive application slowdown caused by the co-run activities. We use performance counter based throttling to mitigate the contention on the memory bandwidth. We evaluate our work on Huawei Mate 7 smartphone with realistic workloads. The result shows significant energy discount (up to 63%) for best-effort tasks with minimum impact on the interactive application execution (3.8% slowdown in the worst case).

Acknowledgments This work was supported in part by the National Science Foundation grants CNS-1217372, CNS-1239423, and CCF-1255729, and by a Google Research Award. We thank Handong Ye, Yi Jiang and Jun Wang from FutureWei Technologies, Inc. and Vijayakumar Krishnamurthy, Anthony Mazzola, Chuk Orakwue from Huawei Device for valuable discussions and support. We also thank the anonymous USENIX ATC reviewers and our shepherd Rodrigo Fonseca for comments that helped improve this paper.

References

[1] Fast, affordable eye tracking. `http://www.sticky.ad/`.

[2] Gamebench setup. `https://www.gamebench.net/en/gamebench-setup`.

[3] iOS 9, Siri. `http://www.apple.com/ios/whats-new/#siri`.

[4] Monsoon power meter. https://www.msoon.com/LabEquipment/PowerMonitor/.

[5] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *Proc. of the USENIX Annual Technical Conf.* (Boston, MA, June 2010).

[6] CARROLL, A., AND HEISER, G. Mobile multicores: use them or waste them. In *Proc. of the Workshop on Power-Aware Computing and System (HotPower)* (Nov. 2013).

[7] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. A non-work-conserving operating system scheduler for SMT processors. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture* (2006).

[8] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the 16th IEEE Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)* (Brasov, Romania, Sept. 2007), pp. 25–38.

[9] FEDOROVA, A., SMALL, C., NUSSBAUM, D., AND SELTZER, M. Chip multithreading systems need a new operating system scheduler. In *Proc. of the SIGOPS European Workshop* (Leuven, Belgium, Sept. 2004).

[10] GAO, C., GUTIERREZ, A., RAJAN, M., DRESLINSKI, R. G., MUDGE, T., AND WU, C.-J. A study of mobile device utilization. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)* (Philadelphia, PA, Mar. 2015).

[11] GOMEZ, L., NEAMTIU, I., AZIM, T., AND MILLSTEIN, T. RERAN: Timing-and touch-sensitive record and replay for Android. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)* (San Francisco, CA, May 2013), pp. 72–81.

[12] GUTIERREZ, A., DRESLINSKI, R. G., WENISCH, T. F., MUDGE, T., SAIDI, A., EMMONS, C., AND PAVER, N. Full-system analysis and characterization of interactive smartphone applications. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (2011), pp. 81–90.

[13] IYER, S., AND DRUSCHEL, P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001), pp. 117–130.

[14] LANE, N. D., CHON, Y., ZHOU, L., ZHANG, Y., LI, F., KIM, D., DING, G., ZHAO, F., AND CHA, H. Piggyback crowdsensing (PCS): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proc. of the 11th ACM Conf. on Embedded Networked Sensor Systems (SenSys)* (Rome, Italy, Nov. 2013).

[15] LUMB, C. R., SCHINDLER, J., GANGER, G. R., AND NAGLE, D. F. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proc. of the 4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Oct. 2000).

[16] MARTINS, M., CAPPOS, J., AND FONSECA, R. Selectively taming background android apps to improve battery lifetime. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 563–575.

[17] MILBORROW, S., AND NICOLLS, F. Active shape models with SIFT descriptors and MARS. In *Proc. of the Int'l Conf. on Computer Vision Theory and Applications (VISAPP)* (Lisbon, Portugal, Jan. 2014), pp. 380–387.

[18] NIKZAD, N., CHIPARA, O., AND GRISWOLD, W. G. APE: An annotation language and middleware for energy-efficient mobile application development. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)* (Hyderabad, India, June 2014), pp. 515–526.

[19] SEO, W., IM, D., CHOI, J., AND HUH, J. Big or little: A study of mobile interactive applications on an asymmetric multi-core platform. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (Atlanta, GA, Oct. 2015).

[20] SHEN, K., SHRIRAMAN, A., DWARKADAS, S., ZHANG, X., AND CHEN, Z. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013).

[21] SHINGARI, D., ARUNKUMAR, A., AND WU, C.-J. Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (Atlanta, GA, Oct. 2015), pp. 22–33.

[22] SONG, W., SUNG, N., CHUN, B.-G., AND KIM, J. Reducing energy consumption of smartphones

using user-perceived response time analysis. In *Proc. of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile)* (Santa Barbara, CA, Feb. 2014).

[23] TAM, D., AZIMI, R., SOARES, L., AND STUMM, M. Managing shared L2 caches on multicore systems in software. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture* (San Diego, CA, June 2007).

[24] UNIFIED EXTENSIBLE FIRMWARE INTERFACE FORUM. Advanced configuration and power interface specification, July 2014. Revision 5.1.

[25] YAN, T., CHU, D., GANESAN, D., KANSAL, A., AND LIU, J. Fast app launching for mobile devices using predictive user context. In *Proc. of the 10th Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys)* (Lake District, United Kingdom, June 2012), pp. 113–126.

[26] YOON, C., KIM, D., JUNG, W., KANG, C., AND CHA, H. AppScope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *Proc. of the USENIX Annual Technical Conf.* (Boston, MA, June 2012).

[27] ZHANG, X., DWARKADAS, S., FOLKMANIS, G., AND SHEN, K. Processor hardware counter statistics as a first-class system resource. In *Proc. of the 11th Workshop on Hot Topics in Operating Systems (HotOS)* (San Diego, CA, May 2007).

[28] ZHANG, X., DWARKADAS, S., AND SHEN, K. Towards practical page coloring-based multi-core cache management. In *Proc. of the 4th EuroSys Conf.* (Nuremberg, Germany, Apr. 2009), pp. 89–102.

[29] ZHU, Q., ZHU, M., WU, B., SHEN, X., SHEN, K., AND WANG, Z. Software engagement with sleeping CPUs. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015).

Beam: Ending Monolithic Applications for Connected Devices

Chenguang Shen (UCLA)* *Rayman Preet Singh* (Samsung Research)*
Amar Phanishayee *Aman Kansal* *Ratul Mahajan*
Microsoft Research

Abstract— The proliferation of connected sensing devices (or *Internet of Things*) can in theory enable a range of applications that make rich inferences about users and their environment. But in practice developing such applications today is arduous because they must implement all data sensing and inference logic, even as devices move or are temporarily disconnected. We develop Beam, a framework that simplifies IoT applications by letting them specify "what should be sensed or inferred," without worrying about "how it is sensed or inferred." Beam introduces the key abstraction of an *inference graph* to decouple applications from the mechanics of sensing and drawing inferences. The inference graph allows Beam to address three important challenges: (1) device selection in heterogeneous environments, (2) efficient resource usage, and (3) handling device disconnections. Using Beam we develop two diverse applications that use several different types of devices and show that their implementations required up to 12× fewer source lines of code while resulting in up to 3× higher inference accuracy.

1 Introduction

Connected sensing devices, such as cameras, thermostats, in-home motion, door-window, energy, water sensors [2], collectively dubbed as the *Internet of Things* (IoT), are rapidly permeating our living environments [3], with an estimated 50 billion such devices in use by 2020 [34]. In theory, they enable a wide variety of applications spanning security, efficiency, healthcare, and others. But in practice, developing IoT applications is arduous because the tight coupling of applications to specific hardware requires each application to implement the data collection logic from these devices and the logic to draw inferences about the environment or the user.

Unfortunately, this monolithic approach where applications are tightly coupled to the hardware, is limiting in two important ways. First, for application developers, this complicates the development process, and hinders broad distribution of their applications because the cost of deploying their specific hardware limits user adoption. Second, for end users, each sensing device they install is limited to a small set of applications, even though the hardware capabilities may be useful for a broader set of applications. How do we break free from this monolithic and restrictive setting? Can we enable applications to be programmed to work seamlessly in heterogeneous environments with different types of connected sensors and devices, while leveraging devices that may only be available opportunistically, such as smartphones and tablets?

To address this problem, we start from an insight that many inferences required by applications can be drawn using multiple types of connected devices. For instance, home occupancy can be inferred by either detecting motion or recognizing people in images, with data sampled from motion sensors (such as those in security systems or Nest [12]), cameras (e.g. Dropcam [4], Simplicam [18]), microphone, smartphone GPS, or using a combination of these sensors, since each may have different sources of errors. *We posit that inference logic, traditionally left up to applications, ought to be abstracted out as a system service, thus decoupling "what is sensed and inferred" from "how it is sensed and inferred"*. Such decoupling enables applications to work in heterogeneous environments with different sensing devices while at the same time benefiting from shared and well trained inferences. Consequently, there are three key challenges in designing such a service:

Device selection: The service must be able to select the appropriate devices in a deployment that can satisfy an application's inference request (including inference accuracy). Device selection helps applications to run in heterogeneous deployments. It also helps applications to operate in settings with user mobility where the set of usable devices may change over time. Moreover, applications can leverage multiple available devices to improve inference accuracy, as shown in Figure 1.

Efficiency: For inferences that are computationally ex-

*Work done during an internship at Microsoft Research

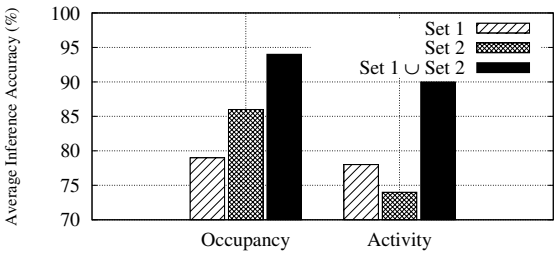


Figure 1: Improvement in occupancy and activity inference accuracy by combining multiple devices in a lab deployment. For occupancy, sensor set 1 = {camera, microphone} in one room and set 2 ={PC interactivity detection} in a second room. For physical activity, set 1 = {phone accelerometer} and set 2 = {wrist worn FitBit [5]}.

pensive to run locally on user devices, or to support deployments that span geographical boundaries, the service should be able to offload computation to remote servers. In doing so, the service should partition computation while efficiently using network bandwidth.

Disconnection tolerance: The service should be able to handle dynamics that can arise due to device disconnections and user mobility.

To address these challenges concretely, we propose *Beam*, an application framework and associated runtime which provides applications with inference-based programming abstractions. It introduces the key abstraction of an **inference graph** to not only decouple applications from the mechanics of sensing and drawing inferences, but also directly aid in addressing the challenges identified above. Applications simply specify their inference requirements, while the Beam runtime bears the onus of identifying the required sensors in the given deployment and constructing an appropriate inference graph.

Inference graphs are made up of modules which are processing units that encapsulate inference algorithms; modules can use the output of other modules for their processing logic. Beam introduces three simple building blocks that are key to constructing and maintaining the inference graph: typed inference data units (IDUs) which guide module composability, channels that abstract all inter-module communications, and coverage tags that aid in device selection. The Beam runtime instantiates the inference graph by selecting suitable devices and assigning computational hosts for each module. Beam also mutates this assignment by partitioning the graph at runtime for efficient resource usage. Beam's abstractions and runtime together provide disconnection tolerance.

Our implementation of the Beam runtime works across Windows PCs, tablets, and phones. Using the framework, we develop two realistic applications, eight different types of inference modules, and add native support for many different types of sensors. Further, Beam supports all device abstractions provided by HomeOS [33], thus enabling the development of a variety of inference modules. We find that for these applications: 1) using Beam's abstractions results in up to 4.5× fewer development tasks and 12× fewer source lines of code with negligible runtime overhead; 2) inference accuracy is 3× higher due to Beam's ability to select devices in the presence of user mobility; and 3) network resource usage due to Beam's dynamic graph partitioning matches hand-optimized versions for the applications.

2 Beam Overview

In this section, we first describe two representative classes of applications and distill the challenges an inference framework should address. Next, we describe the key abstractions central to Beam's design in addressing the identified challenges.

2.1 Example Applications

Our motivation for designing Beam are data-driven-inference based applications, aimed at homes [12, 19], individual users [11, 14, 59, 69, 72] and enterprises [8, 16, 24, 46, 60]. We identify the challenges of building an inference framework by analyzing two popular application classes in detail, one that infers environmental attributes and another that senses an individual user.

Rules: A large class of popular applications is based on the 'If This Then That (IFTTT)' pattern [9, 67]. IFTTT enables users to create their own rules connecting sensed attributes to desired actions. We consider a particular rules application which alerts a user if a high risk appliance, e.g., electric oven, is left on when the home is unoccupied [64]. This application uses the appliance-state and home occupancy inferences.

Quantified Self (QS) [11, 14, 23, 35, 53] disaggregates a user's daily routine by tracking her physical activity (walking, running, etc), social interactions (loneliness), mood (bored, focused), computer use, and more.

Using these two popular classes of applications we address three important challenges they pose: device selection, efficiency, and disconnection tolerance, as detailed in Section 1. Next, we explain the key abstractions in Beam aimed at tackling these challenges.

2.2 Beam Abstractions

In Beam, *application developers* only specify their desired inferences. To satisfy the request, Beam bears the onus of identifying the required sensors and inference algorithms in the given deployment and constructing an inference graph.

Inference Graphs are directed acyclic graphs that connect devices to applications. The nodes in this graph correspond to *inference modules* and edges correspond to *channels* that facilitate the transmission of *inference data units (IDUs)* between modules. While these abstractions

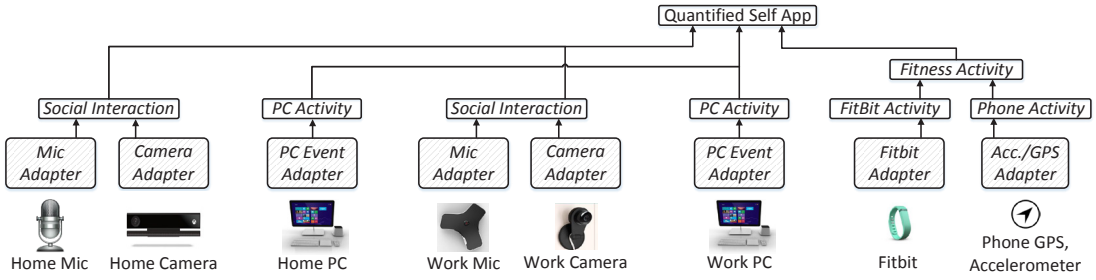


Figure 2: Inference graph of modules for the Quantified Self (QS) app. Adapters are device driver modules.

are described in more detail below, Figure 2 shows an example inference graph for the QS application that we later build and evaluate. The graph uses eight different devices spread across the user's home and workplace, and includes mobile and wearable devices. The application requests a top-level inference as an IDU and Beam dynamically selects the modules that can satisfy this inference based on the devices available. For example, in Figure 2, to satisfy the application's request for inferences pertaining to fitness activities Beam uses a module that combines inferences drawn separately from a user's smartphone GPS, accelerometer, and Fitbit device, thus forming part of the inference graph for QS. Figure 3 shows the inference graph for the Rules application.

Composing an inference as a directed graph enables sharing of data processing modules across applications and other modules that require the same input. In Beam, each computing device associated with a user, such as a tablet, phone, PC, or home hub, has a part of the runtime, called the **Engine**. Engines are computational hosts for inference graphs. Figure 4 shows two engines, one on the user's home hub and another on her phone; the inference graph for QS (shown in Figure 2) is split across these engines, while the QS application runs on a cloud server. For simplicity, we do not show another engine that may run on the user's work PC.

IDU: An *Inference data unit (IDU)* is a typed inference, and in its general form is a tuple <t,e,s>, which denotes any inference with state information *s*, generated by an inference algorithm at time *t* and error *e*. The types of the inference state *s*, and error *e*, are specific to the inference at hand. For instance, *s* may be of a numerical type such as a double (e.g., inferred energy consumption), or an enumerated type such as high, medium, or low. Similarly, error *e* may specify a confidence measure (e.g., standard deviation), probability distribution, or error margin (e.g., radius). IDUs abstract away "what is inferred" from "how it is inferred". The latter is handled by inference modules, which we describe next.

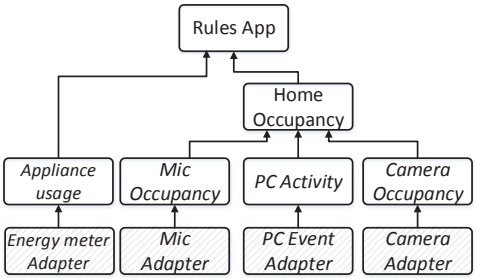


Figure 3: Inference graph for the Rules application.

Inference Modules: Beam encapsulates inference algorithms into modules. Inference modules consume IDUs from one or more modules, perform certain computation using IDU data and pertinent in-memory state, and output IDUs. Special modules called *adapters* interface with underlying sensors and output sensor data as IDUs. Adapters are device drivers that decouple "what is sensed" from "how it is sensed". *Inference developers* specify (i) a module's input dependencies (either as IDU types or as modules), (ii) the IDU type it generates, and (iii) its configuration parameters. Modules have complete autonomy over how and when to output an IDU, and can maintain arbitrary internal states. Listing 1 shows a specification for the Home Occupancy inference module in the Rules inference graph (Figure 3). It lists (i) input dependencies of PC Activity *OR* Mic Occupancy *OR* Camera Occupancy, (ii) *HomeOccupancyIDU* to be the type of output it generates, and (iii) a control parameter, *sampleSize*, that specifies the temporal size of input samples (in seconds) to consider in the inference logic. Application developers request the local engine for desired infer-

```
<Spec>
 <ControlParameters> <!-- Module parameters -->
  <Param name="sampleSize" type="int" value="5"/>
 </ControlParameters>

 <Output> <!-- Output channel IDU spec -->
  <Inference type="Beam.IDU.HomeOccupancyIDU"/>
 </Output>

 <Input> <!-- Input channels -->
  <InputBlock type="OR">
   <InputChannel Mode="FreshPush">
    <Module type="Beam.Modules.PCActivity"/>
    <Module type="Beam.Modules.MicOccupancy"/>
    <Module type="Beam.Modules.CameraOccupancy"/>
   </InputChannel>
  </InputBlock>
 </Input>
</Spec>
```

Listing 1: Module specification of Home Occupancy.

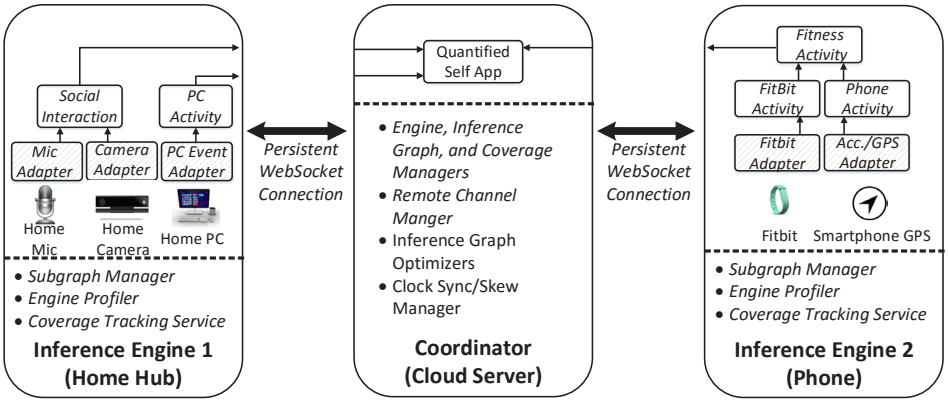


Figure 4: An overview of different components in an example Beam deployment with 2 Engines.

ences, for example:

```
engineInstance.Request(
    Beam.Modules.ModHomeOccupancy,
    tags, Mode.FreshPush);
```

These are satisfied by inference modules implemented by inference developers, and applications receive IDUs via a callback.

Channels: To ease inference composition, *channels* link modules to each other and to applications, abstracting away the complexities of connecting modules across different devices. Channels provide support for disconnections tolerance and enable optimizations such as batching IDU transfers for efficiency. Every channel has a single *writer* and a single *reader* module. Modules can have multiple input and output channels. Channels connecting modules on the same engine are *local*. Channels connecting modules on two different engines, across a local or wide area network, are *remote* channels. Remote channels enable applications and inference modules to seamlessly use remote devices or modules. Channels can be either configured to deliver IDUs to the reader as soon as the writer pushes it (*FreshPush*, as seen in Listing 1 line 12), or to deliver IDUs in batches thus amortizing the cost of computation and network transfers.

Coverage Tags: Coverage tags help manage sensor coverage. Each adapter is associated with a set of coverage tags which describes what the sensor is sensing. For example, a location string tag can indicate a coverage area such as "home" and a remote monitoring application can use this tag to request an occupancy inference for this coverage area. Coverage tags are strongly typed. Beam uses tag types only to differentiate tags and does not dictate tag semantics. This gives applications complete flexibility in defining new tag types. Adapters are assigned tags by the respective engines at setup time, and are updated at runtime to handle dynamics (Section 3.1).

Beam's runtime also consists of a **Coordinator** which interfaces with all engines in a deployment and runs on a replicated server that is reachable from all engines. The coordinator maintains remote channel buffers to support reader or writer disconnections (typical for mobile devices). It also provides a place to reliably store state of inference graphs at runtime while being resistant to engine crashes and disconnections. The coordinator is also used to maintain reference time across all engines. Engines interface with the coordinator using a persistent web-socket connection, and instantiate and manage the parts of inference graphs local to them.

3 Beam Runtime

In this section, we describe how the Beam runtime uses the inference graph to aid in device selection, efficient graph partitioning, and handling device disconnections.

3.1 Device Selection

Beam simplifies application development by automatically selecting devices that match its inference request in heterogeneous deployments and in the presence of user mobility. Beam leverages the device discovery mechanism in HomesOS [33] to discover and instantiate adapter modules for available sensors in the deployment.

Applications request their local Beam engines for all inferences they require, including the coverage associated with each inference. All application requests are forwarded to the coordinator. Using inference module specifications and devices with matching coverage tags available in the deployment [1], the coordinator recursively resolves all required inputs of each module. A module's coverage tag set includes tags from the downstream modules it processes data from.

Handling environmental dynamics: Movement of users and devices can change the set of sensors and devices that satisfy an application's requirement. For instance, consider an application that requires camera input from the device currently facing the user at any time, such as the camera on her home PC, work PC, or smartphone. In such scenarios, the inference graph needs to

[1] The requested tag must match one of the adapter tags.

be updated dynamically. Beam updates the coverage tags to handle such dynamics. Tags of *location* type (e.g., "home") are assumed to be static and are only edited by the user. For tags of type *user*, the sensed subject is mobile and hence the sensors that cover it may change. The coordinator's *tracking service* manages the coverage tags associated with adapters on various engines.

The user tracking service updates the coverage tags as the user moves. When a user leaves home for work, the tracking service removes the *user* tag from device adapters on the home PC and adds them to adapters on her smartphone. When she arrives at work, the tracking service removes the user tag from her smartphone and add them to adapters on her work PC. The user tracking service relies on device interactions. When a user interacts with a device, it updates the tags of all sensors on the device to include the user's tag.

Finally, changes in coverage tags (e.g., due to user movements) or device availability (e.g., device disconnections and re-connections) will result in the coordinator reselecting devices for requested inferences and recreating the graph accordingly.

3.2 Inference Partitioning for Efficiency

Beam uses the inference graph for partitioning computation across devices and optimizing for efficiency.

Graph creation and partitioning: The Beam coordinator maintains a set of inference graphs in memory as an *incarnation*. When handling an inference request, the coordinator first incorporates the requested inference graph into the incarnation, re-using already running modules, and merges inference graphs if needed. Once the coordinator finishes resolving all required inputs for each module in the inference graph, it determines where each module should run using the optimization schemes described next. The coordinator then initializes remote channels and partitions the graph into engine-specific subgraphs which are sent to the engines. Whenever the tracking service updates coverage tags, e.g. due to user movements, the coordinator re-computes the inference graphs and sends updated subgraphs to the affected engines. Next, the engines receive their respective subgraphs, compare each subgraph to existing ones, and update them by terminating deleted channels and modules before initializing new ones. Engines ensure that exactly one inference module of each type with a given coverage tag is created.

Optimizing resource usage: In Beam, optimizations are either performed *reactively*, i.e., when an application issues/cancels an inference request, or *proactively* at periodic intervals.

Beam's default reactive optimization determines where each module should run by partitioning the inference graph to minimize the number of remote channels. Let $G(V,E)$ be an inference graph, where V represents the nodes (inference modules), and E represents its adjacency matrix. In E, e_{ij} is the cost of the edge (channel) connecting module i to module j; $e_{ij} = 0$ if two modules are not connected directly. Beam's optimizer determines potential partitions of the inference graph and picks the partition with the minimum cost. To determine a partition $P_{|V|\times|D|}$, Beam assigns each module $i \in V$ to run on a device $d \in D$. That is, $p_{id} = 1$ if module i runs on device d and $p_{id} = 0$ otherwise. We define the cost matrix of a partition P of the inference graph as $C_{|D|\times|D|} = P^T E P$, where $c_{d_1 d_2}$ denotes the sum of the cost of all channels from device d_1 to device d_2. Since the reactive optimizer aims at minimizing the number of remote channels, here $e_{ij} = 1$ for all connected modules i and j in the graph. An adapter module runs on a device co-located with the sensor, and an application runs on the device requested by the user. Beam solves the following linear program to find P with the minimum cost:

$$\begin{aligned} \text{Minimize} \quad & \sum_{\forall d_1, d_2 \in D, d_1 \neq d_2} c_{d_1 d_2} \\ \text{subject to} \quad & \sum_{d \in D} p_{id} = 1 \qquad \forall i \in V \\ & p_{id} \in \{0,1\} \qquad \forall i \in V,\ \forall d \in D \end{aligned}$$

Beam's default proactive optimization minimizes the amount of data transferred over remote channels by solving the same linear program but using the data rate profile of each edge as e_{ij}. Engines profile their subgraphs, and report profiling data (e.g., per-channel data rate or estimated per-module CPU utilization) to the coordinator periodically. Other potential optimizations can minimize CPU/memory usage at engines, or IDU delivery latency. Beam allows for modular replacement of optimizers. The coordinator applies optimizations by re-configuring inference graphs and remapping the engine on which each inference module runs.

Scatter node optimization: The coordinator further optimizes the inference graph by finding remote channels which have the same writer module, and whose readers reside on a common engine (R_e). For each such set of edges (E), it adds a single remote channel edge from the writer to a new *scatter* node at R_e. The scatter node is then set as the writer for all edges in E, in effect, replacing multiple remote channels with one and reducing the amount of *wide-area* network transfers by a factor of $|E|$.

3.3 Disconnection Tolerance

Beam's remote channels always go through the coordinator and support reader/writer disconnections by using buffers at the coordinator. Thus, a channel is split into three logical components: writer-side, reader-side, and coordinator-side (present only in remote channels). A channel's writer-side and coordinator-side component buffer IDUs. Channels offer two guarantees: i) readers do

Adapter	Inference Module
PC Event	PC Activity
PC Input	PC Occupancy
Phone GPS	Semantic Location
Accelerometer	Fitness Activity
Fitbit	
Energy Meter (HomeOS)	Appliance Usage
Camera (HomeOS)	Camera Occupancy
PC Mic/Tablet Mic	Mic Occupancy
PC Mic/Tablet Mic	Social Interaction

Table 1: Sample adapters and inference modules.

not receive duplicate IDUs, and ii) readers receive IDUs in FIFO timestamp order. Beam specifies a default size for remote channel buffers but also allows application developers to customize buffer sizes based on deployment scenarios, e.g., network delays and robustness.

Internally, channels assign sequence numbers to IDUs. They are used for reader-writer flow control, and in remote channels for applying back-pressure on the writer-side component when the coordinator-side buffer is full, e.g., when a reader is disconnected. Currently, the writer-side and coordinator-side buffers use the drop-tail policy to minimize data transfer from writer to coordinator in the event of a disconnected/lazy reader (as opposed to drop head). This design implies that after a long disconnection a reader will first receive old inference values followed by recent ones.

Channels and modules do not persist data. If necessary, applications and modules may use a temporal data store, such as Bolt [37], to make inferences durable.

4 Implementation

Our Beam prototype is implemented in C# as a cross-platform portable service that can be used by .NET v4.5, Windows Store 8.1, and Windows Phone 8.1 applications. The Beam inference library has sample implementations for 8 inference modules and 9 adapters (listed in Table 1). It also includes a HomeOS-adapter that allows Beam to leverage various other device abstractions provided by HomeOS [33], such as the camera and energy meter device drivers used by some of our sample inferences. Each Beam module has a single data event queue and a thread to deliver received IDUs (akin to the actor model [22, 26, 29]). All communication between the coordinator and engine instances uses the SignalR [17] library, and Json.NET [10] is used for data serialization. The engine library, coordinator, sample adapters, and tracking service are implemented in 6614, 952, 1824, and 219 (total=9609) source lines of code respectively.

4.1 Sample Applications

We implement the motivating applications described in Section 2.1 in Beam. Inference graphs of Rules and Quantified Self (QS) are shown in Figure 3 and Figure 2, respectively. Device adapters such as Microphone, Camera, and PC Event adapters are shared by both inference graphs. For common inference modules such as the PC Activity inference, Beam instantiates only one of them across these graphs. Changes in coverage tags and device availability caused by user mobility prompt Beam to re-select appropriate devices for inference graphs. For instance, PC Activity for QS might either be drawn from the home PC or the work PC depending on the user's current location.

4.1.1 Rules Application

The Rules application requires the Appliance Usage and Home Occupancy inferences implemented as follows.

The *Appliance Usage* inference module reads aggregated power consumption of a home from a whole-home power meter, or a utility smart-meter, and disaggregates it to determine the set of appliances that are on at any given instant, using the CO algorithm from [39], configured with 10 commonly owned home appliances [25]. The whole-house power readings are generated using our power-sensor adapter, which interfaces with an Aeon ZWave whole-house meter [1].

The *Mic Occupancy* inference module reads audio samples using the PC Microphone adapter at a sampling rate of 8 kHz (in 4 second frames), and filters out background noise (such as wind, fans, etc.) [38]. If after filtering, the audio sample still indicates sound is present, the inference output is 'occupied'.

The *PC Activity* module infers the current activity a user is performing on a PC (described in Section 4.1.2).

The *Camera Occupancy* module receives streaming video input from an adapter provided by the HomeOS web-cam driver. The input video is of 640×480 resolution and streams at a frame rate of 1 fps. The module compares consecutive frames in the video. If any significant difference indicating possible human movement is detected [28], the inference output is 'occupied'.

The *Home Occupancy* module combines Mic Occupancy, Camera Occupancy, and PC Activity modules, to produce a Home Occupancy inference, outputting 'occupied' if one of the following is true: Mic Occupancy, Camera Occupancy, or PC Activity $\neq$ No activity.

4.1.2 Quantified Self (QS) Application

QS tracks a user's fitness activities, social behaviors, and computing activities on a PC. It is implemented as a Windows Azure web application. Users view plots of their data at leisure on the QS webpage. The inference modules used by this application are described as follows.

The *Social Interaction (Is Alone)* module detects the presence of human voice, outputting 'user not alone' when human voice is present (likely due to conversations with others, though false positives may arise due to TV sounds and background noises). It computes the mel-

```
1 // Inference developers implement module logic
2 public class ModHomeOccupancy:InferenceModuleBase {
3   // Read parameters from the specification XML file
4   public override void Initialize(ModuleSpec spec) {
5     this.paramList = spec.getControlParams();
6     // set state and initialize using parms ...
7   }
8   // Callback to receive IDUs from input channel(s)
9   public override void DataReceived(IChannel channel
        , List<IIDU> inputSignals) {
10    // Compute occupancy based on input
11    HomeOccupancyIDU inferenceResult =
12                computeOccupancy(inputSignals);
13    // Push result IDUs to output channel(s)
14    if (!changedSinceLastPush(inferenceResult))
15      foreach (IChannel ch in outputChannels)
16        ch.Push(inferenceResult);
17  }
18  // ...
19 }
20 // App developer: request inferences from engine
21 public class QSApp : InferenceModuleBase {
22   void startInference() {
23     // Get an instance of the local engine
24     Beam.Engine engine = Beam.Engine.Instance;
25     // Prepare coverage tags
26     List<CoverageTag> tag = new List<CoverageTag>();
27     tag.Add(new PersonCoverageTag("User1"));
28     // Register for inference notifications
29     engine.Request(Beam.Modules.ModHomeOccupancy, tag
          , Mode.FreshPush, this);
30   }
31   // Callback to receive IDUs from input channel(s)
32   public override void DataReceived(IChannel channel
        , List<IIDU> occupancyInferences ) {
33     // Perform actions based on IDUs received ...
34   }
35 }
```

Listing 2: Example usage of the Beam API.

frequency cepstral coefficients (MFCC) [32, 52] over a 200 ms window of the microphone adapter data at 44.1 kHz and uses a decision tree [58] to classify if human voice is present. The module also incorporates movement detection by analyzing video streams from the camera.

The *PC Activity* inference module reads the name of the currently active desktop window from the PC-event adapter using a Win32 system call. It then classifies the name into one of the known PC activity categories (coding, web browsing, social networking, emailing, reading etc.) using a pre-configured mapping. It also infers the psychological state of the user (bored vs. focused) using the features proposed in [51], including window switches, web page switches, time spent browsing Facebook.com, and time spent using e-mail.

The *Fitness Activity* module implements the algorithm from [61] to infer human transportation modes (still, walking, driving) using the phone accelerometer. It also uses the Fitbit [5] API to fetch users' FitBit activity logs, and combines it with accelerometer-based inferences.

4.2 APIs

Listings 1 and 2 show how application and inference developers leverage the Beam APIs using the Home Occupancy inference as an example.

Inference developers provide an XML specification for each inference module (Listing 1) configuring its parameters as well as the input and output channel IDU types. They then implement the module using Beam's APIs (Listing 2, line 1-19) extending the `InferenceModuleBase` helper class. The module is first initialized with control parameters (line 5). It receives inputs in the `DataReceived` callback (line 9), performs the implemented inference logic (line 11), and sends result IDUs to output channels (line 14-16).

Application developers simply request a specific inference module, e.g. Home Occupancy (Listing 2, line 20-35). The application specifies coverage tags (line 26-27), and invokes the local engine's `Request` method (line 29) to register for inference notifications. Beam then instantiates the required inference graph and returns a channel to the application with the requested module as writer. Result IDUs are received by the application via the `DataReceived` callback (line 32).

Application components and their description
Sensor driver: *Handled by M-Hub and Beam* One driver per sensor type.
Inference logic: *Handled by M-Lib and Beam* For each inference an application requires, at least one inference component is needed, e.g., incorporating feature extraction techniques, inference algorithm, learning model, etc.
Parameter tuning: *Simplified by Beam* An application must also incorporate logic to match its inference logic with the underlying sensors (for a range of sensors), e.g. configuring sensor-specific parameters such as sampling rate, frame rate for cameras, sensitivity level for motion sensors, etc.
Cloud service: *Simplified by Beam* Depending on the development approach, an application may require several cloud services, e.g., a storage service for data archival, an execution environment for hosting inference logic, authentication services, etc.
Device disconnection tolerance: *Handled by Beam* Since devices such as smartphones, tablets, may have intermittent connectivity, developers need to appropriately handle disconnections.
User interface (UI): *Simplified by Beam* Typical applications require certain UI components, e.g., to allow configuration of sensors for data collection, or for users to view results.

Table 2: Components of inference-based applications.

5 Evaluation

We evaluate how Beam's inference graph abstraction simplifies application development, benchmark its performance, and evaluate its efficacy in addressing the three key challenges identified in Section 1. Our evaluation uses micro-benchmarks as well as the two motivating applications from Section 4.1.

First, in Section 5.2 we quantify how Beam's abstractions simplify application development and evaluate the overhead of graph creation. Then, in Section 5.3, we evaluate how Beam's device selection in a real-world deployment with user mobility improves inference accuracy. Next, in Section 5.4, we show the impact of Beam's inference graph partitioning to optimize for efficient resource usage. Finally, in Section 5.5 we showcase Beam's ability to handle device disconnections.

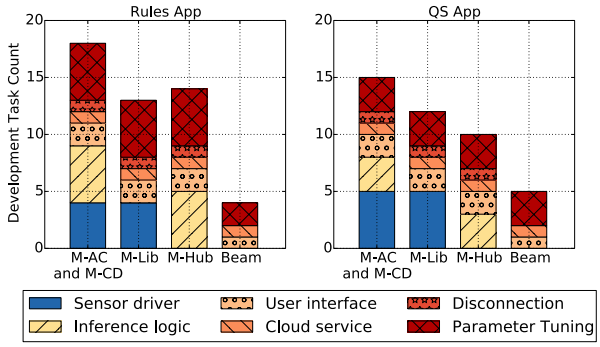


Figure 5: Development tasks using different development approaches in the two applications (Rules, QS)

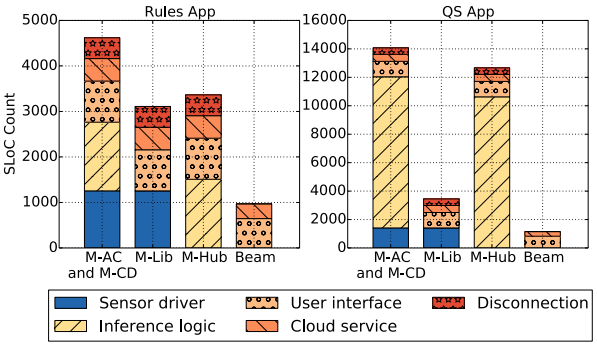


Figure 6: SLoC using different development approaches in the two applications (Rules, QS)

For our experiments, the Beam coordinator runs on a Windows Azure VM (with AMD Opteron Processor, 7 GB RAM, 2 virtual hard disks, running Windows Server 2008 R2); the engines run on desktop machines (with AMD FX-6100 processor, 16 GB RAM, running Windows 8.1) and a Windows Phone (Nokia Lumia). Both sample applications, Rules and Quantified Self (QS), run on the same VM as the coordinator; local engines run in the cloud, a home PC, phone, and a work PC.

5.1 Development Approaches

To quantify the reduction in development effort achieved by Beam, we explore different approaches that a developer may adopt to design such applications.

Monolithic-All Cloud (M-AC). In this approach, the application is developed as a monolithic silo without the use of any framework. All application logic is tightly coupled to the sensing devices, and all collected data is relayed to cloud services, as is the case with Xively [20] and SmartThings [19]. The cloud service runs the application's data processing and inference logic.

Monolithic-Cloud and Device (M-CD). In this approach, an application developer hard-codes the division of inference logic across the cloud VM and end devices [13, 69]. Thus, sensor values are processed to some degree on the end device before being uploaded to the cloud VM which hosts the remainder of the application logic. Depending on the deployment and resource constraints, the developer may need to hand-optimize the resource usage (e.g., CPU, memory, or network usage).

Monolithic-using inference libraries (M-Lib). This approach is similar to the previous one (M-CD), except that application developers may use libraries of inference algorithms tuned by domain experts, thus leading to some reduction in development effort [31, 44, 57].

Monolithic-using sensor hub systems (M-Hub). Platforms such as HomeOS [33], Homeseer [7], and others [15], facilitate the development of applications by providing homogeneous device-based programming abstractions. Typically, these platforms implement sensor drivers and regulate access to different sensors; applications still implement inference logic.

Beam. In this approach, an application on any of the user's devices simply presents its inference requests to the local Beam instance. Using the inference graph abstraction, Beam bears the onus of device selection, optimizing for efficiency, and handling disconnections. Note that using Beam does not preclude the M-Hub approach where all sensing and inference logic run locally on a single hub device (e.g., a home hub). We refer to such scenarios built using Beam's inference abstractions as Beam-Hub, with the engine and coordinator running locally without needing an external network connection.

5.2 Evaluation of Inference Abstraction

In this section we highlight the saving in application development effort using Beam's inference graph abstraction and quantify the overhead of graph creation.

5.2.1 Comparison of Development Effort

We implement our representative applications using the different development approaches described above and present a quantitative comparison of the development effort using two metrics: (i) number of development tasks and (ii) number of source lines of code (SLoC). *Number of development tasks* is defined as the number of architectural components that need to be designed, implemented, and maintained for a complete functioning application. To analyze development effort in greater depth, these components can further be categorized based on the function they perform (Table 2). This metric captures the diverse range of tasks developers of applications for connected devices are required to handle. Although comparing the number of tasks provides insight into the development effort required for each approach, different components often require varying levels of implementation efforts. Thus, to distinguish individual components, we also measure the *number of source lines of code (SLoC)* required for the components in each approach.

Figures 5 and 6 show the number of development tasks and number of SLoC, respectively, for the Rules and QS

	Sample scenario 1 (local inference)		Sample scenario 2 (remote inference)		
	App1's request	App2's request	App1's request	App2's request	Reevaluation
Total	232.54 ± 1.63	246.71 ± 16.62	237.43 ± 12.76	230.24 ± 3.53	-
Request and subgraph transfer	230.35 ± 1.68	246.24 ± 16.62	236.13 ± 12.75	229.73 ± 3.47	-
Coordinator (inference graph creation)	1.05 ± 0.14	0.16 ± 0.01	0.90 ± 0.04	0.20 ± 0.07	0.12 ± 0.01
Coordinator (split inference graphs)	0.06 ± 0.01	0.12 ± 0.01	0.06 ± 0.01	0.15 ± 0.07	0.11 ± 0.01
Engine (instantiate subgraphs)	1.05 ± 0.13	0.16 ± 0.03	0.30 ± 0.08	0.12 ± 0.04	0.40 ± 0.10

Table 3: Inference graph setup times (in ms) in two sample scenarios, with one standard deviation.

applications using the different development approaches. We observe that for the Rules application, Beam reduces the number of development tasks by 4.5×, and the number of SLoC by 4.8×, compared with M-AC and M-CD. Similarly, for the QS application, Beam reduces the number of development tasks by 3×, and the number of SLoC by 12×, compared with M-AC and M-CD.

Number of development tasks: As shown in Figure 5, the approaches of Monolithic-All Cloud (M-AC) and Monolithic-Cloud and Device (M-CD) have similar number of development tasks for both the Rules (on left) and the QS application (on right). M-CD requires developers to hard-code the division of tasks between endpoint devices and cloud servers, thus statically optimizing for better resource usage than M-AC (Section 5.4).

Compared with M-AC and M-CD, the M-Lib approach reduces developer effort. It leverages existing libraries which provide implementations of inference algorithms and also handle their training and tuning. Similarly, in the M-Hub approach, developer effort is reduced due to existing sensor driver implementations provided by the platform. Finally, when using Beam, application developers do not need to design or implement sensor drivers, inference logic, tuning timing parameters, or handling disconnections. Application developers only need to decide their required inferences, and develop application-specific components, e.g., user interface, third-party authentication, etc.

Number of SLoC: As shown in Figure 6, we observe that for all approaches, the SLoC count is generally proportional to the development task count. For most approaches SLoC is dominated by tasks of developing sensor drivers and inference logic. For instance, the Social Interaction inference in QS contributes more than 9796 SLoC. Both Beam and M-Lib help alleviate this complexity. Beam improves upon M-Lib by handling the complexity of implementing sensor drivers, disconnection tolerance, and optimizing resource usage, etc.

5.2.2 Overhead of Inference Graph Creation

We study the time taken by Beam to satisfy requests for a single Mic Occupancy inference, which in turn uses the PC Mic adapter. We consider two sample scenarios, 1) applications request for a local inference, and 2) applications request for a remote inference. In both cases, application 1 initiates a request first, followed by application 2, with the same coverage tag.

In both scenarios, the overhead of instantiating and maintaining the inference graph at end-points is minimal and dwarfed by the latency of transferring the request to the coordinator and receiving back the subgraphs.

Table 3 shows the overhead of graph creation for each of the scenarios. In both cases, the second request uses less time for graph creation at the coordinator, since much of the graph already exists when the second request arrives (e.g., module specifications are not re-read). Likewise, in both scenarios, time spent at the engine(s) in applying the subgraph is lower for the second request as compared to the first request. Further, it is lower in scenario 2 because the inference graph is split across two engines. Lastly, the coordinator performs a periodic re-evaluation based on the channel data rates and applies the proactive optimization discussed in Section 3.2. The time taken to perform the re-evaluation is minimal.

5.3 Device Selection

Unlike other approaches described in Section 5.1, the inference graph in Beam can select devices for applications even in heterogeneous environments with user mobility, resulting in increased inference accuracy. We demonstrate this using the PC Activity inference in the context of the QS application (inference graph in Figure 2).

We perform an experimental lab deployment with two locations - a lab which acts as 'home' and an office. Movements from home to office are used to simulate user commuting. We compute Beam's inference accuracy against manually-collected ground truth data from the deployment, and compare it to three other development approaches that may be used in the absence of a Beam-like tracking service. The first approach performs the PC Activity inference using only inputs from the home PC, while the second approach uses only inputs from the work PC. We assume that the home PC goes into sleep after a certain period of user inactivity, while the work PC remains on even after the user leaves. In the third approach, the inference is drawn using simultaneous inputs from both the home and work PCs. However, when the two inputs conflict, the output is set to 'Other'.

Figure 7 shows a comparison of inference accuracy for these different schemes over a ten minute interval of using the QS application. Inferring PC-based activities using only the home PC works accurately until the user

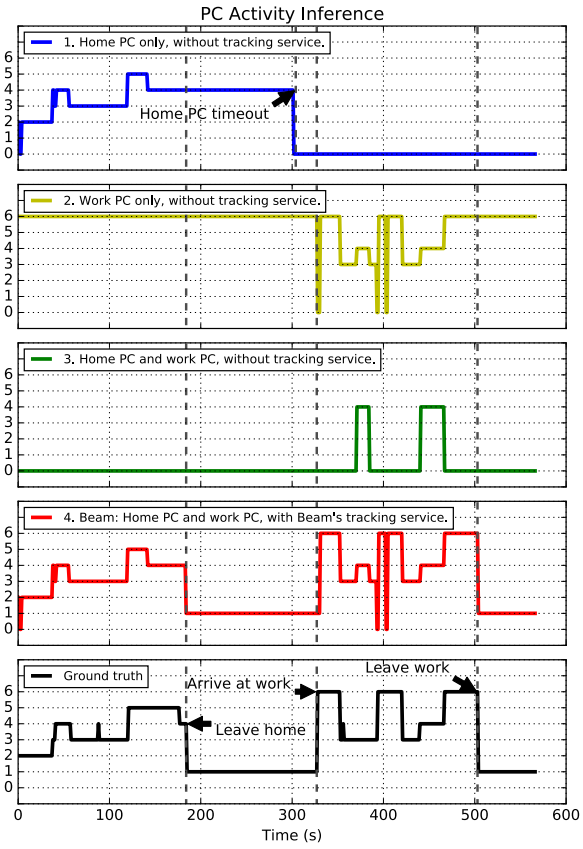


Figure 7: Beam's tracking service improves inference accuracy (measured against ground truth) significantly over other approaches all of which fail to select devices in the presence of user mobility.

leaves home, but deviates significantly from ground truth once the user has left. Similarly, using only the work PC can only accurately compute the PC-based activities of the user after the user arrives at work. On the other hand, using both the work and home PC without a tracking service often produces conflicting results, for instance, when home PC and work PC both generate PC Activity inferences during user commuting. Beam's tracking service correctly identifies the location of the user and triggers the inference graph to re-select appropriate devices, achieving inference accuracy 3× higher than the best performing scheme above. Using the tracking service, Beam's smartphone engine can also correctly indicate that the user is 'Mobile' while commuting. Table 4 summaries these accuracy improvements.

Although the above experiments are performed in a lab setting with a simulated commuting scenario, having a longer commuting time will only reduce the accuracy of non-Beam approaches, since only Beam with the tracking service can infer user commuting and all other approaches will yield incorrect results. Finally, we expect to observe a similar accuracy improvement for other

Setup	Accuracy
Home PC only, without tracking service	29.68%
Work PC only, without tracking service	26.94%
Home PC and work PC, without tracking service	4.59%
Home PC and work PC, with Beam's tracking service	88.16%

Table 4: Accuracy of PC Activity Inference compared to ground truth (a summary of Figure 7).

inferences that require handling of sensor coverage, e.g. the Social Interaction inference in the QS application.

5.4 Efficient Resource Usage

Next, we illustrate that Beam can match the resource usage of hand-optimized applications by partitioning the inference graph across devices. We also evaluate different optimization schemes used in Beam. Although we consider network usage to benchmark Beam in this paper, we expect similar optimizations can be performed on other resources such as CPU usage, latency, and energy.

Graph partitioning: For the Rules and QS applications, we compare Beam's data transfer overhead (i.e., number of bytes transferred over the wide area) with that of different approaches (M-AC, M-CD, M-Lib, M-Hub). Figure 8 shows the total number of bytes transferred over the wide area in one hour, for the sample applications using different approaches. Medians and standard deviations across three runs are reported. M-AC incurs the largest overhead, because it transfers all sensor data from the device to a cloud VM for processing. On the other hand, the M-CD, M-Lib, and M-Hub approaches are optimized to perform most of their processing at the edges before transferring data to the cloud VM. Beam automatically partitions the inference graph using both reactive and proactive optimizations and comes close to matching the network transfer overhead incurred by M-CD; it incurs a slightly higher overhead for transferring *control messages* such as forwarding the application's request to the coordinator, receiving the part of the inference graph to instantiate, sending channel data rates to coordinator (for proactive optimization), acknowledgments, etc. Note that, when the M-Hub approach is used for the Rules application, there is no wide area IDU transfers because all

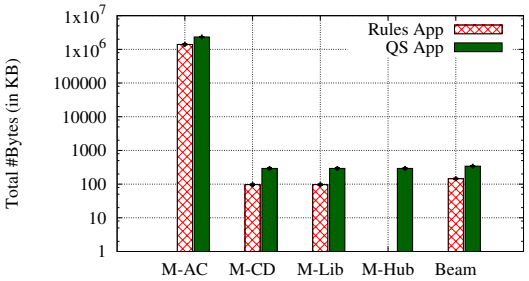


Figure 8: Total bytes transferred over the wide area for a 60 minute run of the Rules and QS apps using different approaches. Y-axis is in log scale.

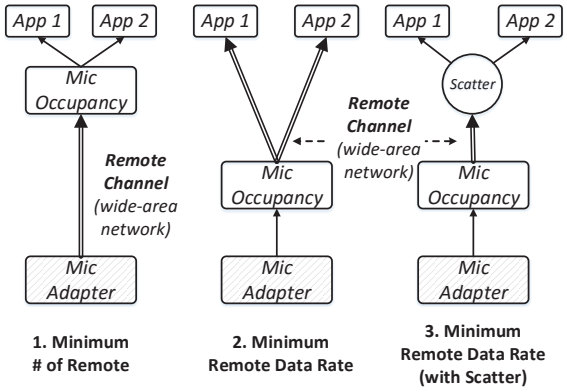


Figure 9: Sample configurations of the Mic Occupancy inference, with different optimization goals.

required sensors are present locally at home.

Optimization schemes: Next, we evaluate the effect of different optimization schemes in Beam. We focus on a simple inference graph, where two applications running on cloud servers subscribe to the Mic Occupancy inference. Figure 9 shows the three configurations that result from Beam optimizations in isolation and Figure 10 shows their network resource consumption over a 100-second interval. Beam's default reactive optimization (Figure 9 #1) minimizes the number of remote channels resulting in a large amount of microphone data being transmitted over the wide area. Beam's proactive optimization notices these large uploads and uses channel data rates to re-evaluate and re-partition the inference graph (Figure 10 at 20 s), thus moving the Mic Adapter closer to the edge (Figure 9 #2), and reducing wide area transfers significantly. Finally, enabling Beam's scatter node optimization (Figure 9 #3) halves the network overhead, for two consumer applications, compared with the proactive optimization without the scatter node.

5.5 Handling Disconnections

In this section we quantify the ability of the Beam inference graph to handle device disconnections. Remote

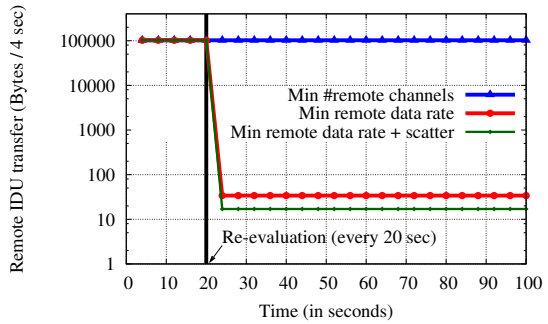


Figure 10: Network resource consumption over a 100 seconds interval for configurations in Figure 9. Y-axis is in log scale. IDUs are generated every 4 seconds.

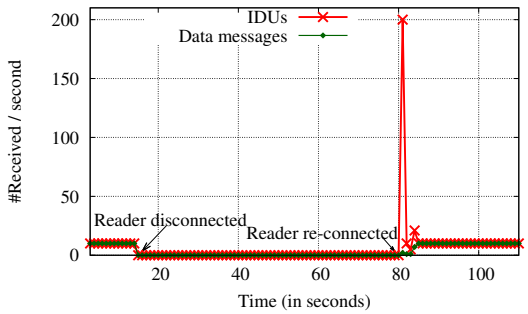


Figure 11: Remote channel time trace. Write rate is 10 values per second, and writer buffer and coordinator buffer are sized at 100 values each.

channels in Beam buffer data at both the coordinator and at the writer endpoints to tolerate reader and writer disconnections. The size of these buffers and the writer's sending rate determine the time window for which disconnections are lossless, and can be sized as per the deployment scenario. Figure 11 shows a time series plot of the number of IDUs and data messages received by a reader over a 100 seconds interval. The writer produces ten IDUs every second. Each IDU produced is pushed out in a separate data message until a reader disconnection at t=15s results in data buffering, first at the coordinator, and then at the writer. We constrain the channel buffers at the writer and coordinator ends to 100 IDUs each, thus supporting buffering of only 20 seconds worth of IDUs in this configuration, forcing the remaining IDUs to be dropped. When the reader reconnects at t=80 s, the 200 buffered IDUs are batched in a small number of data messages and delivered to the reader, showing Beam's support for tolerating device disconnections.

6 Related Work

Beam's inference graph draws inspiration from dataflow graphs used in a wide range of scenarios such as routers [45], operating systems [54, 62], data-parallel computation frameworks [6, 40], and Internet services [70]. Beam is the first framework that provides inference abstractions to decouple applications, inference algorithms, and devices, using the inference graph for device selection, efficiency, and disconnection tolerance. We classify prior work into four categories.

Device abstraction frameworks: HomeOS [33] and other platforms [7, 15, 21, 68] provide homogeneous programming abstractions to communicate with devices. For instance, HomeOS applications can use a generic motion sensor role, regardless of the sensor's vendor and protocol. These approaches only decouple device-specific logic from applications, but are unable to decouple inference algorithms from applications. Moreover, they cannot provide device selection or inference partitioning capabilities.

Cross-device frameworks: Rover [41], an early distributed object programming framework for mobile applications, allows programmers to partition client-server applications; it provides abstractions such as relocatable objects and queued remote procedure calls to ease application development. Sapphire [73], a more recent framework, requires programmers to specify per-object deployment managers which aid in runtime object placement decisions, while abstracting away complexities of inter-object communication. MagnetOS [48] dynamically partitions a set of communicating Java objects in a sensor network with a focus on energy efficiency. Like these frameworks, channels in Beam abstract away local and remote inter-module communication. Beam fundamentally differs from them by using the inference graph to decouple applications from sensing and inferences, aid in device selection to operate in heterogeneous environments, and support global resource optimizations.

Macro-programming frameworks [27, 36, 49] provide abstractions to allow applications to dynamically compose dataflows [50, 56]. Semantic Streams [71] and Task Cruncher [66] address sharing sensor data and processing across devices. However these approaches focus on data streaming and simple processing methods, e.g., aggregations, rather than generic inferences, and do not target general purpose devices e.g., phones, PCs. In addition, they do not address device selection or inference partitioning at runtime.

Mobile sensing frameworks: Existing work has focused only on applications requiring continuous sensing on a *single* mobile device. Kobe [31], Auditeur [57], and Senergy [44] propose libraries of inference algorithms to promote code re-use and explore single device energy-latency-accuracy trade-offs. Other work [42, 43, 44, 47] has focused on improving resource utilization by sharing sensing and processing across multiple applications on a mobile device. None of these approaches address problems such as modular inference composition, device selection with user mobility, inference partitioning across multiple devices, or handling disconnections.

An early version of our work appears in a workshop paper that outlined the problem and presented a basic design [65]. The current paper extends the design, implements real applications, and evaluates performance.

7 Discussion

We discuss potential improvements to Beam.

Error and error propagation: Beam currently supports typed errors such as probability distributions (e.g. mean and standard deviation), and error margin (e.g. center and radius). Although error propagation has been studied in the field of artificial intelligence (e.g. neural network [63]), there is no prior work on error propagation in mobile context sensing. We are investigating techniques to enable inference module developers to implement customized error propagation functions for specific inferences, so that Beam can propagate the error from a module's inputs to its output.

Actuation and timeliness: Many in-home devices possess actuation capabilities, such as locks, switches, cameras, and thermostats. Applications and inference modules in Beam may want to use such devices. If the inference graph for these applications is geo-distributed, timely propagation and delivery of such actuation commands to the devices becomes important and raises interesting questions of what is the *safe* thing to do if an actuation arrives "late".

Data archival and correlation mining: Prior work has shown that exploiting the correlation among inferences can effectively reduce sensing cost [55]. While Beam modules do not currently store data either at the engines or the coordinator, applications and modules may use a temporal datastore, such as Bolt [37], to make inferences durable. Storing and querying archived inference data will allow inference developers to perform correlation mining to improve inferences.

Data privacy: While we do not address privacy concerns in our work, we believe the use of inferences can enable better data privacy controls [30]. For example, users may allow an application to access the occupancy inference (using a camera) instead of the raw image data used for drawing the inference. This prevents the leakage of private information by preventing other inferences that can be drawn using the raw data. Moreover, Beam's coverage tags allow the user to define fine-grained controls, for instance, allowing an application to access activity inference only for a certain user tag.

8 Conclusion

Applications using connected sensing devices are difficult to develop today because they must incorporate all the data sensing and inference logic, even as devices move or are temporarily disconnected. We design and implement Beam, a framework and runtime for distributed applications using connected devices. Beam introduces the inference graph abstraction which is central to decoupling applications, inference algorithms, and devices. Beam uses the inference graph to address the challenges of device selection, efficient resource usage, and device disconnections. Using Beam, we develop two representative applications (Rules and QS), where we show up to 4.5× lower number of tasks and 12× lower source line of code in application development effort, with negligible runtime overhead. Moreover, Beam results in up to 3× higher inference accuracy due to its ability to select devices in heterogeneous environments, and Beam's dynamic optimizations match hand-optimized applications for network resource usage.

References

[1] Aeon Labs Z-Wave Smart Energy Switch. http://aeotec.com/.

[2] Amazon Home Automation Store. http://www.amazon.com/home-automation-smarthome/b?ie=UTF8&node=6563140011.

[3] The US Market for Home Automation and Security Technologies. Technical report, BCC Research, IAS031B, 2011.

[4] Dropcam - Super Simple Video Monitoring and Security. https://www.dropcam.com/.

[5] Fitbit. https://www.fitbit.com/.

[6] Google Cloud Dataflow. https://cloud.google.com/dataflow/.

[7] HomeSeer. http://homeseer.com/.

[8] iOS: Understanding iBeacon. http://support.apple.com/kb/HT6048.

[9] IFTTT: Put the internet to work for you. https://ifttt.com/.

[10] Json.NET. http://www.newtonsoft.com/json.

[11] Map my fitness. http://www.mapmyfitness.com/.

[12] Nest. http://www.nest.com/.

[13] Optimized App. http://optimized-app.com/.

[14] Quantified self. http://quantifiedself.com/.

[15] Revolv. http://revolv.com/.

[16] ShopKick shopping app. http://shopkick.com/.

[17] ASP.NET SignalR. http://signalr.net/.

[18] Simplicam Home Surveillance Camera. https://www.simplicam.com/.

[19] SmartThings. http://www.smartthings.com/.

[20] xively by LogMein. https://xively.com/.

[21] A. Amiri Sani, K. Boos, M. H. Yun, and L. Zhong. Rio: A system solution for sharing I/O between mobile systems. In *Proc. ACM MobiSys*, 2014.

[22] J. Armstrong. Erlang. *CACM*, 53:68–75, Sept 2010.

[23] E. Arroyo, L. Bonanni, and T. Selker. Waterbot: Exploring feedback and persuasive techniques at the sink. In *Proc. ACM CHI*, 2005.

[24] R. K. Balan, K. X. Nguyen, and L. Jiang. Real-time trip information service for a large taxi fleet. In *Proc. 9th ACM MobiSys*, June 2011.

[25] N. Batra, J. Kelly, O. Parson, H. Dutta, W. J. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava. NILMTK: An open source toolkit for non-intrusive load monitoring. In *Proc. ACM e-Energy*, 2014.

[26] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[27] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive Mob. Comput.*, 3(4):386–412, Aug. 2007.

[28] A. B. Brush, J. Jung, R. Mahajan, and F. Martinez. Digital neighborhood watch: Investigating the sharing of camera data amongst neighbors. In *Proc. ACM CSCW*, 2013.

[29] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *Proc. ACM SOCC*, 2011.

[30] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava. ipShield: A framework for enforcing context-aware privacy. In *Proc. 11th USENIX NSDI*, April 2014.

[31] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proc. 9th ACM SenSys*, Nov. 2011.

[32] S. B. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 28(4):357–366, 1980.

[33] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[34] D. Evans. The Internet of Things: How the next evolution of the Internet is changing everything. *CISCO white paper*, 2011.

[35] J. Froehlich, L. Findlater, M. Ostergren, S. Ramanathan, J. Peterson, I. Wragg, E. Larson, F. Fu, M. Bai, S. Patel, and J. A. Landay. The design and evaluation of prototype eco-feedback displays for fixture-level water usage data. In *Proc. ACM CHI*, 2012.

[36] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *Distributed Computing in Sensor Systems*, pages 126–140. Springer, 2005.

[37] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[38] T. Hao, G. Xing, and G. Zhou. isleep: Unobtrusive sleep quality monitoring using smartphones. In

Proc 11th ACM SenSys, 2013.

[39] G. Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 1992.

[40] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*, Mar. 2007.

[41] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proc. 15th ACM SOSP*, Dec. 1995.

[42] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. Symphoney: A coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proc. 10th ACM SenSys*, Nov. 2012.

[43] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: Scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proc. ACM MobiSys*, 2008.

[44] A. Kansal, S. Saponas, A. B. Brush, K. S. McKinley, T. Mytkowicz, and R. Ziola. The Latency, Accuracy, and Battery (LAB) abstraction: Programmer productivity and energy efficiency for continuous mobile context sensing. In *Proc. ACM OOPSLA*, Nov. 2013.

[45] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[46] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the North Sea. In *Proc. 3rd ACM SenSys*, Nov. 2005.

[47] F. Lai, S. S. Hasan, A. Laugesen, and O. Chipara. Csense: A stream-processing toolkit for robust and high-rate mobile sensing applications. In *Proc. 13th ACM/IEEE IPSN*, 2014.

[48] H. Liu, T. Roeder, K. Walsh, R. Barr, and E. G. Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *Proc. ACM MobiSys*, 2005.

[49] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 5(3):543–576, 2006.

[50] G. Mainland, M. Welsh, and G. Morrisett. Flask: A language for data-driven sensor network programs. *Harvard Univ., Tech. Rep. TR-13-06*, 2006.

[51] G. Mark, S. T. Iqbal, M. Czerwinski, and P. Johns. Bored Mondays and focused afternoons: The rhythm of attention and online activity in the workplace. In *Proc. 32nd ACM CHI*, April 2014.

[52] P. Mermelstein. Distance measures for speech recognition, psychological and instrumental. *Pattern recognition and artificial intelligence*, 116: 374–388, 1976.

[53] D. Morris, A. B. Brush, and B. R. Meyers. Superbreak: Using interactivity to enhance ergonomic typing breaks. In *Proc. ACM CHI*, 2008.

[54] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. 2nd USENIX OSDI*, Oct. 1996.

[55] S. Nath. Ace: Exploiting correlation for energy-efficient and continuous context sensing. In *Proc. ACM MobiSys*, 2012.

[56] R. Newton, G. Morrisett, and M. Welsh. The Regiment macroprogramming system. In *Proc. 6th ACM/IEEE IPSN*, 2007.

[57] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proc. 11th ACM MobiSys*, June 2013.

[58] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[59] M. M. Rahman, A. A. Ali, K. Plarre, M. al'Absi, E. Ertin, and S. Kumar. mConverse: Inferring conversation episodes from respiratory measurements collected in the field. In *Proc. 2nd Wireless Health*. ACM, 2011.

[60] H. Ramamurthy, B. S. Prabhu, R. Gadh, and A. Madni. Wireless industrial monitoring and control using a smart sensor platform. *Sensors Journal, IEEE*, 7(5):611–618, May 2007.

[61] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava. Using mobile phones to determine transportation modes. *ACM Transactions on Sensor Networks (TOSN)*, 6(2):13, 2010.

[62] D. M. Ritchie. A stream input-output system. In *AT&T Bell Laboratories Technical Journal*, 1984.

[63] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[64] R. P. Singh, S. Keshav, and T. Brecht. A cloud-based consumer-centric architecture for energy data analytics. In *Proc. ACM e-Energy*, 2013.

[65] R. P. Singh, C. Shen, A. Phanishayee, A. Kansal, and R. Mahajan. A case for ending monolithic apps for connected devices. In *Proc. HotOS XV*, May 2015.

[66] A. Tavakoli, A. Kansal, and S. Nath. On-line sensing task optimization for shared sensors. In *Proc. 9th ACM/IEEE IPSN*, 2010.

[67] B. Ur, E. McManus, M. Pak Yong Ho, and M. L. Littman. Practical trigger-action programming in the smart home. In *Proc. ACM CHI*, 2014.

[68] M. Van Kleek, K. Kunze, K. Partridge, et al. Opf: a distributed context-sensing framework for ubiquitous computing environments. In *Ubiquitous Computing Systems*, pages 82–97. Springer, 2006.

[69] R. Wang, F. Chen, Z. Chen, T. Li, G. Harari, S. Tignor, X. Zhou, D. Ben-Zeev, and A. T. Campbell. Studentlife: Assessing behavioral trends, mental well-being and academic performance of college students using smartphones. In *Proc. ACM Ubicomp*, 2014.

[70] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM SOSP*, Oct. 2001.

[71] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Proc. EWSN*, 2006.

[72] D. Wyatt, T. Choudhury, J. Bilmes, and J. A. Kitts. Inferring colocation and conversation networks from privacy-sensitive audio with implications for computational social science. *ACM Trans. Intell. Syst. Technol.*, 2(1), Jan. 2011.

[73] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proc. USENIX OSDI*, 2014.

Caching Doesn't Improve Mobile Web Performance (Much)

Jamshed Vesuna[†] Colin Scott[†] Michael Buettner[◇] Michael Piatek[◇]
Arvind Krishnamurthy[⋆] Scott Shenker[†‡]
[†]UC Berkeley [‡]ICSI [◇]Google [⋆]University of Washington

Abstract

A recent NSDI paper [1] reported that increasing the cache hit ratio for an HTTP proxy from 22% to 32% improved median page load time (PLT) for mobile clients by less than 2%. We argue that there are two main causes for this weak improvement: objects on the critical path are often not cached, and the limited computational power of mobile devices causes computational delays to comprise a large portion of the critical path.

Both of these factors were, in fact, outlined by a previous analysis of desktop web performance [2]. However, we (as the authors of the HTTP proxy [1]) did not properly understand the analysis and could have saved ourselves substantial engineering costs if we had. We therefore argue for the need to highlight this prior analysis, and extend the analysis to include mobile devices with slow CPUs, precise cache hit ratios, and a controlled reproduction of the HTTP proxy caching results [1]. In the extreme case of a perfect cache hit ratio, desktop page load times are improved notably by 34% compared to no caching, but mobile page load times only improve by 13% in the median case. We extract a back-of-envelope performance model from these results to help understand their underlying causes.

1 Introduction

Web caching is widely used to reduce network link utilization, decrease server load and data usage, improve reliability for origin web servers, and improve latency for end hosts. Here, we focus exclusively on web caching's effect on latency, as measured by web page load time.

Flywheel [1], Google's HTTP proxy for mobile devices, increased its overall cache hit ratio from 22% to 32%, yet observed only a 1–2% reduction in page load time in the median case. As the designers of Flywheel, we were initially surprised by this weak improvement. If we had been able to predict that caching would have such negligible effects, we could have saved ourselves substantial engineering costs.

In Sections 5 and 6 of their paper on measuring the critical paths of web page loads [2], Wang et al. seek to demonstrate use cases for their measurement tool. Two of their use cases—an analysis of varying CPU speeds, and an analysis of pages loaded with cold vs. warm vs. hot caches—in fact outline the likely root causes for Flywheel's result.

In this paper we seek to highlight Wang et al.'s analysis, as we suspect that we are not alone in holding the misconception that caching should improve latency. We also extend their analysis along several dimensions. We present a methodology for varying cache hit ratios at fine granularity, and measure caching's effects on web performance of both a mobile device and a desktop browser in a controlled and reproducible manner. In our controlled environment, we reproduce Flywheel's reported cache hit ratio increase for a set of 400 Alexa web pages [3] and find a comparable 1% decrease in PLT in the median case. In the extreme case of a perfect cache hit ratio, we find that desktop page load times are reduced notably by 34% compared to no caching, but mobile page load times are only reduced by 13% in the median case.

We develop a back-of-the-envelope performance model and fit its parameters to empirical observations to better understand the underlying causes. Our model indicates that CPU speed is the key resource bottleneck preventing mobile devices from benefiting significantly from web caching. The analysis by Wang et al. seems to further indicate that objects on the critical path are often not cached (or even cacheable).

Our analysis demonstrates that the generally favorable desktop latency improvements from caching do not carry over well to mobile clients. Content providers may want to think twice about expending resources on caching as a means for improving latency, especially as the volume of mobile traffic begins to overtake desktop traffic.

2 Background & Performance Model

Both browsers and mobile applications typically load content using the HTTP(S) protocol. When a user directs the browser (or application) to a new URL, the browser's Object Loader fetches the root HTML object, as depicted in Figure 1. The HTML Parser launches additional fetch requests for each linked resource within the HTML. In this way, the browser incrementally generates the DOM. As the page loads, the Rendering component paints the UI.

From the user's perspective, the performance of a website can be defined according to a number of different metrics [4,5]. Here, we focus on page load time, which is simple to measure and loosely standardized across browsers [6].

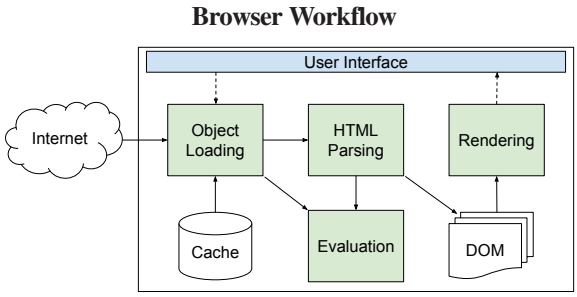


Figure 1: How a browser loads a web page. Reproduced from WProf [2].

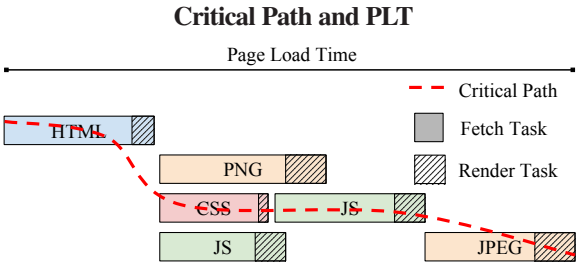


Figure 2: Page load time is determined from the critical path. Objects on the right are dependent on objects to their left, and objects at the same horizontal position are loaded concurrently.

Page Load Time. Roughly speaking, page load time (PLT) is the elapsed time from the moment a user requests a web page to the moment all resources on the page have been loaded [7]. We measure PLT by listening to the browser's JavaScript `onload` event, which fires in most browsers when all resources have been added to the DOM, and all images, scripts, links, and sub-frames have finished loading [6].

Critical Path. Web pages are comprised of many objects, such as images, JavaScript, CSS, and HTML. Each of these objects is handled by multiple (possibly concurrent) browser tasks: it must be fetched, parsed or evaluated, and rendered. We refer to the non-overlapping delays involved in parsing and rendering an object as its 'computational delay,' and refer to the fetch delay as its 'network delay.'

Critical path analysis is a method for analyzing the performance of parallel processes such as browsers. Certain load tasks are dependent on others and must wait until their predecessor tasks have completed. The critical path of a web page is the longest chain of dependent browser tasks such that reducing the length of any task not on the critical path will not change the page load time [8]. In Figure 2, the network and computational delay for the HTML, CSS, JS, and JPEG objects determine the PLT. If we were to decrease the delay for loading the PNG object, the critical path would remain the same, and therefore, the PLT would not change.

2.1 Performance Model

We can now develop an understanding of caching's effect on PLT with a simple performance model. First, consider the following terms:

– Let X denote a given cache hit ratio. We define cache hit ratio as the fraction of all objects in a web page that are served by a cache. Note that the maximum value of X is the fraction of cacheable items on the page, which may be less than 1.

– Let K denote the fraction of objects on the critical path that are cacheable.

– Let N denote the summation of network fetch delays for all objects on the critical path for a cold ($X=0$) page load.

– Let C denote the summation of computational delays for all objects on the critical path for a cold ($X=0$) page load.

– Let $f(X)$ denote the overlap between computational delay and network delay on the critical path. Normally, dependent objects on the critical path should not overlap. There are, however, some cases where the browser can begin concurrently loading an object when its predecessor is only partially loaded. For most purposes, we can ignore $f(X)$.

For simplicity, let us assume that (i) the critical path does not change as we vary the cache hit ratio, (ii) the probability of an object being in cache is uniform across all cacheable objects, and (iii) cached items incur zero network delay. The probability of an object on the critical path incurring a network delay is then:

$$1-Pr(\text{cacheable})\cdot Pr(\text{cache hit}\,|\,\text{cacheable})$$

The expected value of the PLT for a given X is therefore:

$$E_{PLT}[X]=C+(1-K\cdot X)\cdot N-f(X)$$

2.2 Fitting the Model

Sections 5 and 6 of the WProf paper [2] contain empirical measurements of critical paths that allow us to gain a rough understanding of the values of N, C, and K in our model above.

Fitting N and C. In Figure 3, we reproduce WProf's "what-if" analysis (Figure 13 from WProf) for torchbrowser.com. This experiment investigates the performance impact of varying network and computation speeds. We first multiply the computational or network delays for all objects in a web page by a fixed constant. Then, we recompute the page's critical path (based on task dependencies captured by WProf), and extract a predicted PLT. The comp=1 line represents the (2 GHz) desktop CPU that loaded the original page, while comp=0 represents an infinitely fast CPU, comp=1/2 represents a CPU that is twice as fast, and comp=2 (not present in WProf's analysis) represents a CPU of half the speed.

For the infinitely fast CPU (comp=0) we see that its normalized PLT with an unchanged network speed (ratio of network time = 1) should be ~0.8. As we improve network delays for this CPU, we should see a theoretically infinite speedup (tending towards a PLT of 0). Conversely, for the slowest CPU (comp=2), the normalized PLT for an infinitely fast network (ratio of network time = 0) is ~0.4. For this hypothetical CPU (assuming $f(\cdot)$ is close to 0), we can

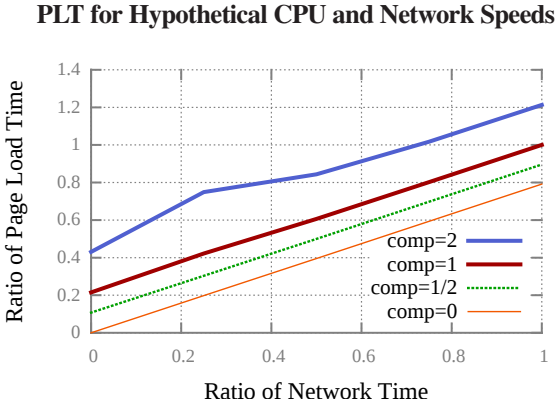


Figure 3: Predicted PLT for torchbrowser.com when hypothetical computation and network speeds are varied.

estimate that the fraction of the critical path that consists of computational delays is ~0.4, while the fraction of the critical path that consists of network delays is ~0.6.

The key takeaway from this analysis is that, as we decrease the speed of the CPU, the ratio of $C:N$ continues to increase. For example, our analysis suggests that a typical mobile device with a ~1 GHz CPU [9] has a $C:N$ ratio of ~2/3 for websites similar to torchbrowser.com. This makes intuitive sense: slower CPUs would cause computational delays to make up larger fractions of the critical path compared to faster CPUs.

Fitting K. To generate Figure 11b in WProf [2], the authors measured the fraction of objects that were in cache immediately after loading pages with a cold cache. Although 65% of all objects were cached, only 20% of all objects on critical paths were cached, giving us an estimate of $K = 0.2$.

Implications. The analysis by WProf, together with our model, give us a rough understanding of the performance effects from caching. We return to our model in §4, where we seek to gain a deeper empirical understanding.

3 Experimental Apparatus

In order to gain a deeper understanding of the performance dynamics outlined in §2.1 and §2.2, we need to experimentally evaluate questions about how caching affects PLT in a controlled environment. Here, we develop our methodology.

Our experimental apparatus (publicly available at [10]) makes use of Telemetry [11] and Web Page Replay (WPR) [12] to measure the effects of parameterized levels of network delays. Both Telemetry and WPR are part of Chromium [13], the open source components of Google Chrome. Here, we use the term 'browser' interchangeably with Google Chrome.

Web Page Replay. WPR acts as a local DNS and HTTP(S) proxy cache (depicted in Figure 4a). In record mode, WPR forwards HTTP(S) requests to the Internet, and records all requests and responses that it observes, as well as metadata such as observed network delays. From this, Web

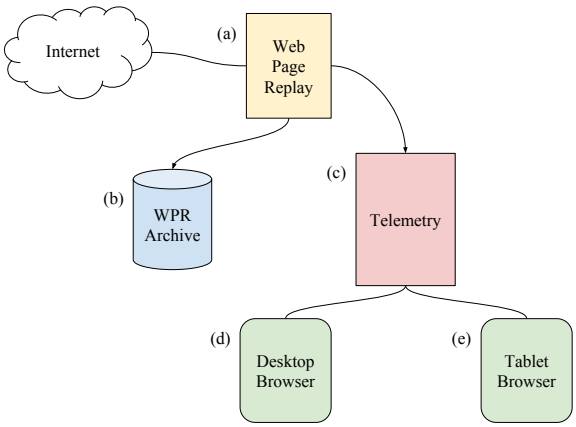


Figure 4: Logical units of our apparatus.

Page Replay builds a WPR archive (depicted in Figure 4b), complete with all HTTP(S) requests, responses, headers, data, and network delays.

In replay mode, the WPR proxy responds to HTTP(S) requests with the responses saved in the archive, or a 404 error response if the corresponding response is not saved in the archive. We configure WPR to send the matching response and data only after sleeping for the time duration originally observed as the network delay between the origin and the WPR proxy (while it was in record mode). Here, WPR is emulating an edge cache rather than a browser cache.

Telemetry. Telemetry (Figure 4c) is a browser performance testing framework, which orchestrates the behavior of the browser and WPR. We use Telemetry to control one of two browsers. The first is a desktop version of Google Chrome running within a virtual machine (specifications in 3.2). The second is a mobile version of Google Chrome running on a USB-tethered mobile device. We use Telemetry to load requested URLs in the browser (as if the user is entering URLs into the omnibox) and passively measuring PLT.

3.1 Workflow

Before each experiment, we clear the browser cache to ensure consistency across trials. For each device (desktop and mobile), we execute the following steps for each URL:

First, we record the live web page from the Internet using Telemetry to instruct the browser to fetch the given URL. The WPR proxy receives this HTTP(S) request, forwards it to the Internet, and passively inspects and records the two-way traffic as noted in §3. We store this data as a WPR archive.

Next, we determine the page load time of the web page with a cold cache. With WPR in replay mode, we load the URL four times (cf. §3.3) and take the minimum page load time as our PLT value, to account for variance.

Now, we emulate a "perfect," fully populated cache. First, we copy the original WPR archive into a new WPR archive.

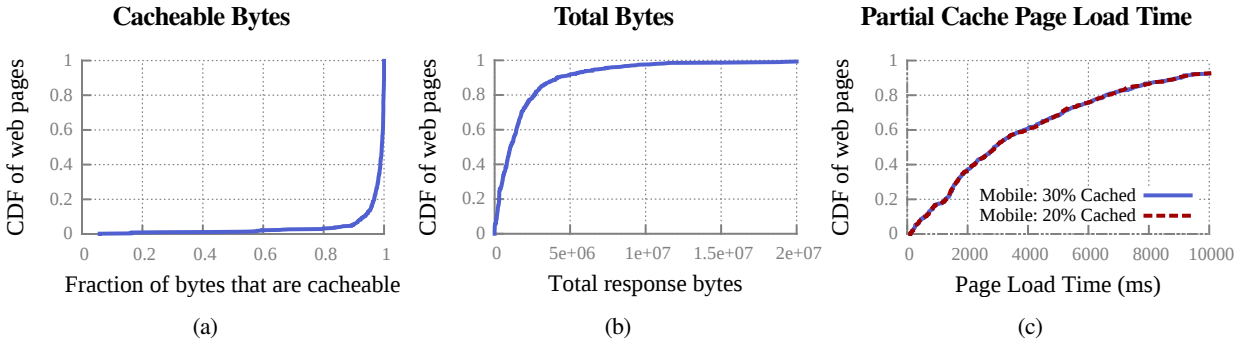


Figure 5: (5a): The majority of web pages are composed mostly of cacheable bytes. (5b): While 95% of web pages are under 6.8 MB, the median web page size is less than 1.2 MB. (5c): Increasing the cache hit ratio from 20% to 30% had negligible effects on mobile PLT.

For each cacheable response in this new WPR archive, we set its network delay to 0 (of course, a "real" cache response time of 0 is not possible, but we set this as an absolute lower bound). Non-cacheable items, as indicated by HTTP headers, retain their initial network delays. We store this modified archive alongside the original (Figure 4b).

Lastly, we record the page load time again, however this time using the modified WPR archive. We determine the PLT in the same way as the original. We then compare the page load times of the unmodified replay executions to that of the modified "perfect cache" executions.

Partial Caching Methodology. To confirm Flywheel's findings, we created two additional sets of partially cached WPR archives: one that caches a randomly chosen set of 30% of all cacheable resources (regardless of byte size), and another that caches 20%. We ensure that the cached items in the 20% WPR archive are a strict subset of the cached items in the 30% WPR archive for consistency.

3.2 Specifications

Each web page was originally fetched over UC Berkeley's LAN, which approximates 250 Mbps down and 230 Mbps up. Our mobile device is a Galaxy Tab 4 with a 1.2 GHz quad-core processor and 1.5 GB on board RAM running Android 4.4, KitKat. Desktop results were performed in a virtual machine with a 3.2 GHz quad-core processor and 6 GB RAM.

3.3 Known Limitations

We identify the following limitations of our apparatus, and discuss the reasoning behind our choice of tools:

Page Load Time as a Metric. When determining web page performance, we chose to focus on page load time rather than SpeedIndex [5] or above-the-fold time [4]. Although they are arguably preferable metrics (as they do a better job of capturing the user's perspective), these metrics are significantly more difficult to measure.

WPR Measurement Accuracy. The PLT measurements taken by WPR are not necessarily consistent with PLTs observed on live web pages, nor are they necessarily consistent across multiple runs of WPR. First, although WPR attempts to mitigate non-determinism in JavaScript execution (by injecting a script into each web page that interposes on non-deterministic calls such as `getTime`), JavaScript may nonetheless exhibit non-determinism across different loads. Second, the mechanism WPR uses to emulate the original RTTs observed during record mode (sleeping a fixed number of milliseconds) may not perfectly match the behavior of the original page load. We try to mitigate these artifacts by loading each web page four times and taking the minimum PLT.[1]

4 Results

Here, we demonstrate empirical performance results we have found with our apparatus. We also attempt to highlight the underlying effects that determine our results.

4.1 Workload Characteristics

We first note several key characteristics of our data corpus:

Data Set. We selected a random subset of 400 of the Alexa top 2000 URLs [3] and loaded their root URL ('/').

Fraction of Cacheable Bytes. Over 90% of web pages in our workload have more than 90% of their total bytes marked as cacheable, as shown in Figure 5a.

Total Bytes. Figure 5b shows the spread of web page sizes in our data set. While 95% of web pages are under 6.8 MB, the median web page size is less than 1.2 MB.

Initial Network Delays. Across all requests/response pairs, the median delay between sending the request and receiving the first response byte was 50ms, with a mean of 151.17ms and standard deviation of 403.77ms.

[1] We observed that beyond four loads per web page, the minimum PLT value did not decrease significantly.

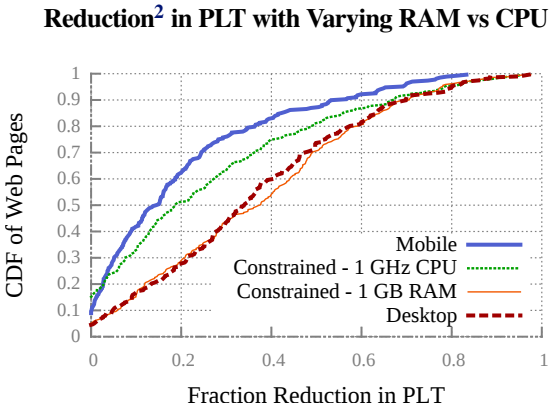


Figure 6: Reduction in page load time due to (perfect) caching is significantly less on mobile devices than desktop. Further, CPU, not RAM, is the primary resource that differentiates mobile devices from their desktop counterparts.

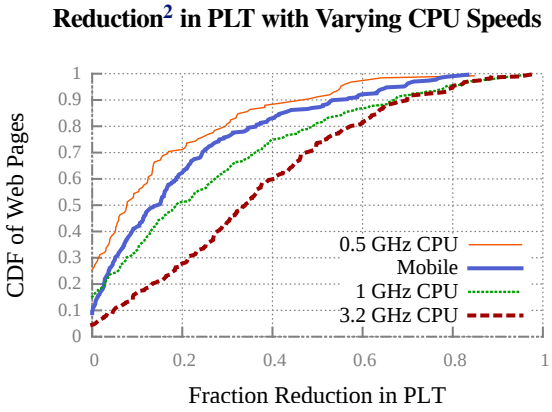


Figure 7: Slower CPU speeds cause increasingly diminished benefits from (perfect) caching.

User Agent. Many web pages are now optimized for mobile devices. Web servers inspect the user agents (UA) of incoming HTTP(S) requests to deliver customized content to the client depending on their device size and computational resources. We ran all of our experiments twice: once where the browsers (both desktop and mobile) advertised a mobile UA, and once where the browsers advertised a desktop UA. We found that the differences in the results were comparable. Here, we show only the desktop UA results to make our graphs more readable.

4.2 Performance Results

As we saw in Figure 5a, 90% of pages are composed of >90% cacheable bytes. If network delays were dominant and the fraction of cacheable objects on the critical path were moderately high, one would conclude that PLT would become negligible with a perfect cache. As our model predicts however, this is not the observed outcome.

Caching Doesn't Significantly Reduce Mobile PLT. We reproduced Flywheel's result in our controlled environment by emulating cache hit ratios of 20% and 30%. As shown in Figure 5c, we found that increasing the hit ratio by 10 percentage points had negligible effects on mobile page load time. Consistent with the reported Flywheel result, we observed only a 1% reduction in PLT in the median case. With a *perfect* cache (Figure 6), our mobile device gains only a 13% PLT reduction in the median case, while its desktop counterpart sees a PLT reduction of 34%.

Limited RAM Does Not Affect Computational Delays. It is possible that either limited RAM or limited CPU would increase computational delays on the critical path. Here, we seek to isolate which of these resources plays a larger role.

A typical mobile device in the global market today has a 1 GHz processor and 1 GB of RAM [9]. We emulate these conditions and isolate computational resources with virtual machines that were constrained by different resources (using VirtualBox to either set a limit on memory usage or emulate a slower CPU clock speed). With 'Mobile' and (unconstrained) 'Desktop' as baselines, Figure 6 presents a stark contrast between these resource constraints: our CPU constrained VM ('Constrained - 1 GHz CPU') behaves very similarly to our tablet ('Mobile'), while our RAM constrained VM ('Constrained - 1 GB RAM') is more closely aligned with the 'Desktop' results. From this we conclude that CPU, not RAM, plays the larger role in determining the performance improvements from caching.

Slow CPU Speeds Increase Computational Delays. Now that we have isolated CPU as the bottleneck resource, we seek to measure the extent of its impact. In terms of our model, we already know that slower CPUs should incur higher computational delays, but here we seek to understand the empirically observed ratios of $C:N$ (as opposed to the hypothetical, predicted ratios in Figure 3). In Figure 7, we observe that, as predicted, as we throttle CPU constraints, perfect caching has noticeably smaller effects on PLT.

The Marginal Benefits of Caching Sharply Decrease. Figure 8 illustrates that for each 10 percentage point increase in cache hit ratio, there is only a 1 percentage point decrease in mobile page load time. That is, there is a sharply diminishing marginal performance gain for every additional byte that is cached. This experimental evidence supports our model: although we do not directly measure the critical path (since WProf is not currently available for mobile browsers), it appears that the fraction of cacheable bytes on the critical path (K) is significantly smaller than the fraction of cacheable byes *not* on the critical path.

4.3 Data Validation

We made several efforts to sanity check our results [14]. To mitigate non-determinism, we compared the status codes

[2]The fraction reduction in PLT for a web page is defined as (Original PLT - PLT with a perfect cache) / (Original PLT).

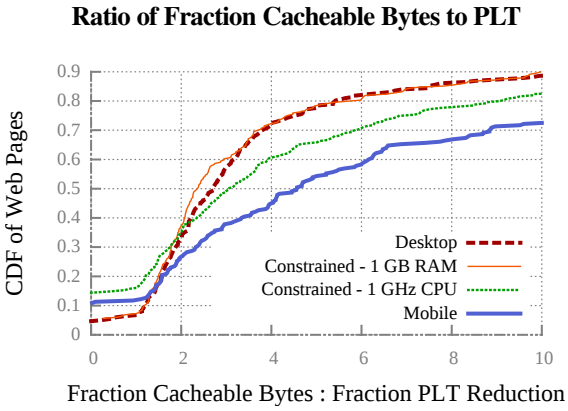


Figure 8: As the percentage of cacheable bytes in a web page increases, the reduction in page load time due to caching increases. However, for each additional percentage cached, there is less than a percentage reduction in page load time.

of all objects loaded in the browser from both original and perfect/partial cache benchmarks. We filtered out about 9% of web pages in our 400 URL data set in cases where there were a high number of 404 error codes due to non-deterministic requests without responses in the WPR archive. The figures we present show only these 91% of web pages that passed our non-determinism filter.

As the ratio of cached to non-cached bytes increases in a web page, we expect page load time to be less than or equal to that of its non-cached counterpart. As seen in Figure 8, there is a positive correlation between the fraction of cached bytes and the reduction in PLT, albeit asymptotic. However, due to variance in PLT (discussed in §3.3), we see in Figure 6 that ~10% of web pages perform worse when cached, as indicated by the data points to the left of `X = 0`.

5 Related Work

Several papers have analyzed web performance, caching, and the relationship between CPU speeds and PLT.

WProf. Wang et al. [2] is the closest research to ours. As we discussed in §2, the experiments WProf ran for their Figure 11 show that objects on the critical path are often not cached; and the experiments they ran for their Figure 13 implies that decreasing CPU speed causes computational delays to comprise a larger fraction of the critical path.

We extend their research along several dimensions. We develop a model that allows us to predict PLT for a given cache hit ratio. We show that limited RAM does not increase computational delays, though slow CPUs do. We also empirically measure (rather than statically compute, as WProf does) PLTs using a tablet device, and using CPU-constrained virtual machines, over a larger data set (400 URLs, vs. ~50 URLs). Lastly, we extend WProf's cacheability analysis to show that the marginal returns from caching sharply diminish.

Concurrently with our work, Nejati et al. ported WProf to mobile browsers [15]. Although they do not consider caching, their tool would be invaluable for deepening our analysis.

Web Performance. Related studies [16, 17] focus on evaluating and optimizing web performance for desktops. Many techniques such as altering content, data compression, proxy services, and CDNs have been exploited to reduce latency for users. These studies focus on high performing end devices such as desktops. We additionally analyzed and compared web performance on a mobile device.

Zhen Wang et al. [18, 19] have determined that the largest delay factor in desktop web page loading is object rendering in the browser. They went further to show that CPU constraints are the lead cause of slow resource loading. With a large data set, we bolster their claim that CPU constraints are the critical factor in determining page load time. We also demonstrate that web caching has diminishing benefits due to the limited CPU speeds of mobile devices.

We are not the first to focus on web performance for mobile devices [18, 19]. Our main contribution is developing a performance model for pinpointing the key differences between desktop and mobile.

Web Caching. Other literature [20–23] has focused on the benefits of web caching, specifically the reduction of latency for desktops [24–28]. While these papers make note of the several benefits of caching, they do not focus on highlighting caching's effects on (CPU-constrained) mobile devices.

Proposed Changes to the Web. There are many papers [29–37] that propose changes to the web that would improve web latency with better caching schemes. It is possible that under their proposed changes, caching would have more of a benefit for mobile latency. In this paper, we focus only on today's existing infrastructure.

6 Conclusion

Motivated by our initial surprise at Flywheel's weak performance improvements from smarter caching, we sought to highlight and extend the analysis done by Wang et al. [2], which indicates two reasons caching should not significantly improve page load time for mobile devices: slow CPU speeds, and sparsity of cached items on the critical path. To make effective use of caching, content providers should pay careful attention to whether cached objects are on the critical path.

Going forward, mobile devices are becoming increasingly powerful, and the bottleneck resources will shift. We hope that the model we have developed here will help content providers and network designers make informed decisions about the performance effects of caching for the mobile web of the future.

Acknowledgments. We thank the anonymous reviewers for their feedback, and especially our shepherd Dan Tsafrir for helping us develop our performance model. This research was supported by an NSF Graduate Research Fellowship.

References

[1] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15, 2015.

[2] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying Page Load Performance with WProf. NSDI '13, 2013.

[3] Alexa Internet Inc. Alexa Top 500 Global Sites. http://www.alexa.com/topsites, 2015. Accessed: 2015-9-12.

[4] Pat Meenan Jake Brutlag, Zoe Abrams. Above the Fold Time: Measuring Web Page Performance Visually. O'Reilly Media, Inc., 2011. Accessed: 2016-1-19.

[5] Google. Speed Index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[6] W3C. Navigation Timing Level 2 Spec. http://w3c.github.io/navigation-timing/.

[7] Google Developers. PageSpeed Insights, 2015. Accessed: 2016-1-11.

[8] Vivek Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical report, Stanford Univ., CA (USA), 1987.

[9] Statista. Global market share held by leading smartphone vendors from 4th quarter 2009 to 3rd quarter 2015, 2015. Accessed: 2016-1-26.

[10] Jamshed Vesuna. Telemetry, Web Page Replay Experimental Apparatus. https://github.com/JamshedVesuna/telemetry.

[11] Google. Telemetry. https://catapult.gsrc.io/telemetry.

[12] Google. Web Page Replay. https://github.com/chromium/web-page-replay.

[13] Google Chromium. Google. https://www.chromium.org/.

[14] Jamshed Vesuna. Sanity Checks. https://github.com/colin-scott/page_load_time/tree/master/telemetry/sanity_checks.

[15] Javad Nejati and Aruna Balasubramanian. An In-Depth Study of Mobile Browser Performance. WWW, 2016.

[16] Steve Souders. High-performance web sites. *Communications of the ACM*, 2008.

[17] Patrick Killelea. *Web Performance Tuning: Speeding up the Web*. " O'Reilly Media, Inc.", 2002.

[18] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. Why are Web Browsers Slow on Smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM, 2011.

[19] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of the 21st international conference on World Wide Web*. ACM, 2012.

[20] Bernhard Ager, Fabian Schneider, Juhoon Kim, and Anja Feldmann. Revisiting Cacheability in Times of User Generated Content. In *INFOCOM IEEE Conference on Computer Communications Workshops, 2010*. IEEE, 2010.

[21] Jia Wang. A Survey of Web Caching Schemes for the Internet. *ACM SIGCOMM Computer Communication Review*, 1999.

[22] Lothar Braun, Alexander Klein, Georg Carle, Helmut Reiser, and Jochen Eisl. Analyzing Caching Benefits for YouTube Traffic in Edge Networks A Measurement-based Evaluation. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*. IEEE, 2012.

[23] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Usenix symposium on internet technologies and systems*, 1997.

[24] David A Patterson. Latency Lags Bandwith. *Communications of the ACM*, 2004.

[25] Armando Fox and Eric A Brewer. Reducing WWW Latency and Bandwidth Requirements by Real-time Distillation. *Computer Networks and ISDN Systems*, 28, 1996.

[26] Kun-Lung Wu and S Yu Philip. Latency-sensitive Hashing for Collaborative Web Caching. *Computer Networks*, 2000.

[27] Pablo Rodriguez, Keith W Ross, and Ernst W Biersack. Improving the WWW: Caching or Multicast? *Computer Networks and ISDN Systems*, 1998.

[28] Swaminathan Sivasubramanian, Guillaume Pierre, Maarten van Steen, and Gustavo Alonso. Analysis of Caching and Replication Strategies for Web Applications. *Internet Computing, IEEE*, 2007.

[29] Leo A Meyerovich and Rastislav Bodik. Fast and Parallel Webpage Layout. In *Proceedings of the 19th international conference on World wide web*. ACM, 2010.

[30] Jeffrey Erman, Alexandre Gerber, Mohammad T Hajiaghayi, Dan Pei, and Oliver Spatscheck. Network-aware Forward Caching. In *Proceedings of the 18th International Conference on World Wide Web*. ACM, 2009.

[31] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. Smart Caching for Web Browsers. In *Proceedings of the 19th international conference on World wide web*. ACM, 2010.

[32] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. In *ACM SIGCOMM Computer Communication Review*, volume 28. ACM, 1998.

[33] John Dilley and Martin Arlitt. Improving Proxy Cache Performance: Analysis of Three Replacement Policies. *IEEE Internet Computing*, 1999.

[34] Guohong Cao. A Scalable Low-latency Cache Invalidation Strategy for Mobile Environments. *Knowledge and Data Engineering, IEEE Transactions on*, 2003.

[35] Guohong Cao. Proactive Power-aware Cache Management for Mobile Computing Systems. *Computers, IEEE Transactions on*, 2002.

[36] Sunho Lim, Wang-Chien Lee, Guohong Cao, and Chita R Das. A Novel Caching Scheme for Improving Internet-based Mobile Ad Hoc Networks Performance. *Ad Hoc Networks*, 4, 2006.

[37] Chanda Dharap. Semantics-based Caching Policy to Minimize Latency, 1999. US Patent App. 09/374,694.

Secure and Efficient Application Monitoring and Replication

Stijn Volckaert[*†] *Bart Coppens*[†] *Alexios Voulimeneas*[*] *Andrei Homescu*[‡]
Per Larsen[*‡] *Bjorn De Sutter*[†] *Michael Franz*[*]

[*]*UC Irvine* [†]*Ghent University* [‡]*Immunant, Inc.*

Abstract

Memory corruption vulnerabilities remain a grave threat to systems software written in C/C++. Current best practices dictate compiling programs with exploit mitigations such as stack canaries, address space layout randomization, and control-flow integrity. However, adversaries quickly find ways to circumvent such mitigations, sometimes even before these mitigations are widely deployed.

In this paper, we focus on an "orthogonal" defense that *amplifies* the effectiveness of traditional exploit mitigations. The key idea is to create multiple *diversified* replicas of a vulnerable program and then execute these replicas in lockstep on identical inputs while simultaneously monitoring their behavior. A malicious input that causes the diversified replicas to diverge in their behavior will be detected by the monitor; this allows discovery of previously unknown attacks such as zero-day exploits.

So far, such *multi-variant execution environments* (MVEEs) have been held back by substantial runtime overheads. This paper presents a new design, ReMon, that is non-intrusive, secure, and highly efficient. Whereas previous schemes either monitor every system call or none at all, our system enforces cross-checking only for security critical system calls while supporting more relaxed monitoring policies for system calls that are not security critical. We achieve this by splitting the monitoring and replication logic into an in-process component and a cross-process component. Our evaluation shows that ReMon offers same level of security as conservative MVEEs and run realistic server benchmarks at near-native speeds.

1 Motivation

Low-level memory errors can lead to reliability and security problems in systems software implemented in C/C++. In principle, we can eliminate such errors by enforcing spatial and temporal memory safety properties at run time [28, 29]. However, the resulting performance overheads prohibit widespread deployment of such solutions in practice [39].

The ubiquity of multi-core CPUs makes multi-variant execution environments (MVEEs) increasingly attractive to improve the reliability and security of code likely to contain memory corruption vulnerabilities [7, 8, 12, 16, 17, 26, 35, 40, 20]. The idea is to monitor the execution of multiple diversified program replicas for divergence in their observable behavior when an exploit triggers implementation-specific, unintended behavior [21].

Security-oriented MVEEs execute replicas in lockstep and typically perform monitoring at a system call granularity, suspending replicas before system calls and checking their arguments for equivalence. In case of divergence, execution is terminated to limit the effects of an attack. Such MVEEs use operating system processes for isolation between the replicas and the host system and between the replicas and the monitoring component as shown in Figure 1(a). Unfortunately, cross-process monitoring (CP) designs incur substantial performance overheads due to frequent context switching and the resulting translation-lookaside buffer (TLB) and cache flushes.

Hosek et al. [17] developed an alternative reliability-oriented MVEE, VARAN, using in-process (IP) rather than CP monitoring (see Figure 1(b)). VARAN outperforms CP monitors by removing the need for context switching and trades lockstep execution for a loosely synchronized execution model. VARAN, however, does not protect the host system from compromised replicas and is therefore less suitable for security-oriented use cases.

This paper proposes a new, hybrid MVEE design—*ReMon*—that uses an existing, isolated CP monitor (GHUMVEE [42]) to enforce lockstep execution for all sensitive system calls. To increase efficiency, we augment GHUMVEE with a compact, security-hardened IP monitor (IP-MON) that enables efficient replication of non-sensitive calls without context switching. As a result, our design (see Figure 1(c)) unites the strengths of the previous approaches. It provides security guarantees that are comparable to those of existing security-oriented MVEEs while approaching the efficiency of VARAN.

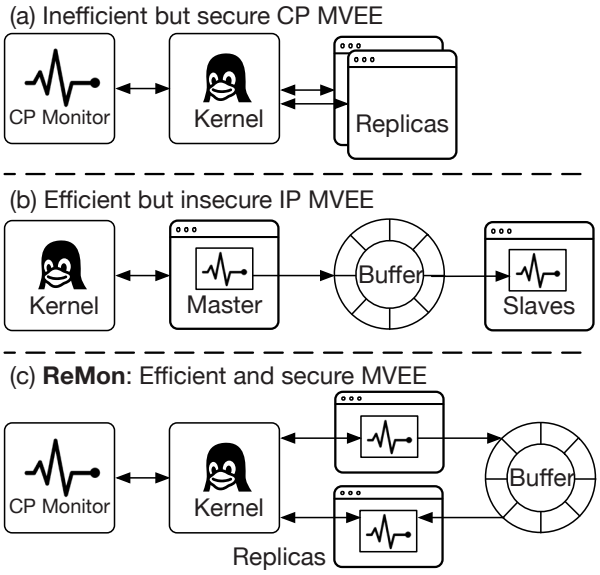


Figure 1: Three MVEE designs running two replicas.

Our design is motivated by the fact that a security policy of monitoring *all* system calls is overly conservative [14, 32]. Many system calls cannot affect any state outside of the process making the system call. Only a small set of *sensitive* system calls are potentially useful to an attacker. Thanks to the IP-MON component, ReMon supports configurable *relaxation* policies that allow non-sensitive calls to execute without being cross-checked against other replicas. Section 5 evaluates the performance impact of a range of relaxation policies inspired by the classification of system calls used in OpenBSD [25].

In summary, our paper contributes:

- **A Novel MVEE Design** ReMon unifies the strengths of previous approaches: the security properties of traditional cross-process MVEEs [8, 12, 35, 40], and the efficient replication mechanism of in-process reliability-oriented MVEEs [17].
- **Relaxed Monitoring Techniques** We leverage our split-monitor design to support relaxed monitoring policies. The IP-MON component lets replicas make certain system calls without cross-process monitoring to increase efficiency.
- **Extensive Evaluation & High Performance** We implemented a full-fledged prototype of our ReMon design and perform a careful and detailed evaluation under different relaxed monitoring policies. Our evaluation shows that ReMon compares favorably to previous work and allows server applications to run in lockstep at near-native speeds.

2 Background

Several MVEE designs have been explored in the past decade. Broadly speaking, two key factors distinguish them. First, an MVEE can run entirely in kernel space [12], or run in user space. Second, some MVEEs execute within the context of the replicas' processes [12, 17], whereas other run in a separate process [10, 35, 40]. These designs make different trade-offs. In-kernel designs are problematic from a security standpoint because MVEE monitors typically have a large attack surface and are prone to memory corruption themselves. The large attack surface arises from the need for the monitor to interpose on every system call executed by a replica, whereas the possibility of memory corruption is due to the plethora of specialized functions that compare and copy complex data structures, such as I/O and message vectors. Successful attacks on MVEE monitors cannot be ruled out, and, in the case of in-kernel monitors, they could easily compromise the entire system.

User-space designs, on the other hand, contain the damage that an attacker can inflict in case the monitor is compromised. Some designs place the MVEE monitor inside the replicas' processes (in-process monitoring / IP), whereas the majority of designs isolate the monitoring process (cross-process monitoring / CP). An IP implementation as shown in Figure 1(b) allows monitor-replica interaction without context switching, but lacks a hardware-enforced protection boundary to isolate the replicas from the monitor. Misbehaving replicas might therefore interfere with the monitor, unless they are augmented with Control-Flow Integrity (CFI) [1] or Software-based Fault Isolation (SFI) [43]. CFI and SFI would, however, reduce or negate the performance benefits of IP monitoring.

In contrast, a CP MVEE (Figure 1(a)) does not require program transformations that slow down the replicas throughout the entire execution. Interaction between a CP MVEE and its replicas does require context switching, however, which is a costly operation due to the need to switch page tables and flush the TLB. When implementing a security-oriented MVEE, the choice between an IP or CP monitoring design is ultimately a trade-off between efficient interaction between the monitor and the replicas (IP design) or faster execution of the replicas (CP design).

2.1 Transparent I/O Replication

The MVEE's monitor ensures that the replicated execution is transparent to the end user. Apart from timing, an outside observer should not notice any differences between native execution of a single replica and Multi-Variant Execution of multiple replicas. The MVEE therefore guarantees that externally observable I/O operations execute only once, while at the same time ensuring that all replicas receive consistent I/O results.

ReMon handles this transparent I/O replication using a master/slave model, similarly to several existing MVEEs. One replica is designated as the master; all other replicas become slaves. Whenever replicas invoke an I/O-related system call, ReMon allows only the master to complete the call. When the call in the master has returned, the

system call results are copied to the slaves' memory and all replicas are resumed.

This mechanism also ensures that all replicas receive consistent input. Consistency and transparency are not identical concerns. Many system calls, e.g., those that query the state of a process, do not have effects that are observable to the end user, but they might still return different results if the monitor does not intervene.

Many programs communicate with other processes over shared physical memory pages. This is generally not safe in an MVEE, however, because (i) it prevents the MVEE from providing consistent input to all replicas, since shared memory can be accessed without system calls, and (ii) shared memory allows for unmonitored bi-directional communication channels between replicas. Such communication channels are a challenge to all security-oriented MVEEs. The ability for the replicas to communicate freely increases the likelihood that attackers can mount an asymmetrical attack, in which they provide different inputs to different replicas.

Security-oriented MVEEs, including ours, therefore typically impose restrictions on the use of shared memory. ReMon rejects any request to set up shared memory pages that can form a bi-directional channel. Typically, this restriction does not break programs, because nearly all of them fall back to alternative communication mechanisms when their requests to map shared memory get rejected. We refer to earlier work for a discussion on solutions to support programs that do not have such a fall-back mechanism [10].

2.2 Consistent Signal Delivery

Whereas synchronous signals (such as SIGSEGV) as a direct result of the executing instruction streams can safely be delivered to all replicas, asynchronously delivered signals can cause the replicas to diverge if their corresponding signal handlers are not invoked at the same point in their execution. Most MVEEs therefore defer the delivery of asynchronous signals until all replicas are suspended in equivalent states. ReMon implements the same strategy. It uses the `ptrace` API to discard signals when they are initially delivered and to re-initiate delivery once all replicas have reached a synchronization point.

2.3 Multi-Threaded Replicas

Non-determinism is a common problem in multi-threaded replicas. Non-deterministic replicas might execute different system call sequences, even if they are given the same inputs and if related system calls are prevented from interleaving. To resolve this, ReMon embeds a small Record/Replay agent in each replica to force them to execute user-space synchronization operations in the same order, thereby enforcing equivalent behavior in all replicas. We refer to the literature for an extended discussion on non-determinism in multi-threaded programs as well as available solutions [4, 6, 13, 22, 30, 33, 34].

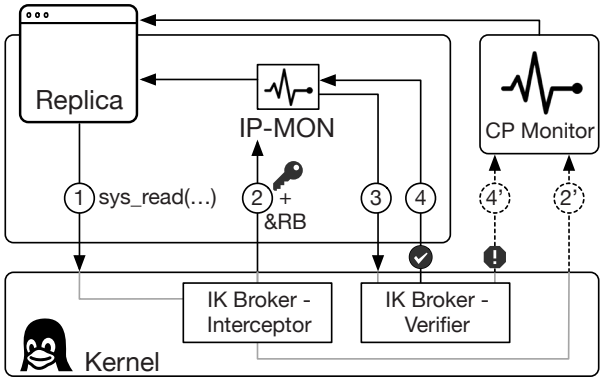


Figure 2: ReMon's major components and interactions.

3 ReMon Design and Implementation

ReMon supervises the execution of an arbitrary number of diversified program replicas that run in parallel. ReMon's main goals are (i) to monitor all of the security-sensitive system calls—hereafter referred to as "monitored calls"—issued by these replicas, (ii) to force monitored calls to execute in lockstep, (iii) to disable monitoring and lockstepping for non-security-sensitive system calls—hereafter referred to as "unmonitored calls", thus allowing the replicas to execute these calls as efficiently as possible while still providing them with consistent system call results, and (iv) to support configurable monitoring relaxation policies that define which subset of all system calls are considered non-security-sensitive, and should therefore not be monitored. ReMon[1] uses three main components to attain these goals:

1. **GHUMVEE** A security-oriented CP monitor implemented as discussed in Section 2. Although GHUMVEE can be used standalone, it only handles monitored calls when used as part of ReMon.

2. **IP-MON** An in-process monitor loaded into each replica as a shared library. IP-MON provides the application with the necessary functionality to replicate the results of unmonitored calls.

3. **IK-B** A small in-kernel broker that forwards unmonitored calls to IP-MON and monitored calls to GHUMVEE. IK-B also enforces security restrictions on IP-MON, and provides auxiliary functionality that cannot be implemented in user space. The broker is aware of the system calls that IP-MON handles and of the relaxation policy that is in effect.

These three components interact whenever a replica executes a system call, as shown in Figure 2.

Our kernel-space system call broker, IK-B, intercepts the system call ① and either forwards it to IP-MON ②, or

[1] `https://github.com/stijn-volckaert/ReMon`

to GHUMVEE ②'. The call is forwarded to IP-MON only if the replica has loaded an IP-MON that can replicate the results of the call, and if the active relaxation policy allows the invoked call to be executed as an unmonitored call. If these two criteria are not met, IK-B uses the standard `ptrace` facilities to forward the call to GHUMVEE instead, which handles it exactly as a regular CP-MVEE.

In the former case, IK-B forwards the call by overwriting the program counter so that the system call returns to a known "system call entry point" in IP-MON's executable code. While doing so, IK-B gives IP-MON a one-time authorization to complete the execution of the call without having the call reported to GHUMVEE. The broker grants this authorization by passing a random 64-bit token ② as an implicit argument to the forwarded call. IP-MON then performs a series of security checks and eventually completes the execution of the forwarded call by restarting it ③. IP-MON can choose to restart the call with or without the authorization token still intact. If the token is intact upon reentering the kernel, IK-B allows the execution of the system call to complete, and returns the call's results to IP-MON ④. If the token is not intact, or if IP-MON executes a different system call, or if the first system call executed after a token has been granted does not originate from within IP-MON itself, IK-B revokes the token and force the call to be forwarded to GHUMVEE ④'.

IP-MON generally executes unmonitored system calls only in the master replica, and replicates the results of the system call to the slave replicas through the replication buffer (RB) discussed in Section 3.2. The slaves wait for the master to complete its system call and copy the replicated results from the RB when they become available.

Although IP-MON allows the master replica to run ahead of the slaves, it still checks if the replicas have diverged. To do so, the master's IP-MON deep copies all of its system call arguments into the RB, and the slaves' IP-MONs compare their own arguments with the recorded ones when they invoke IP-MON. This measure minimizes opportunities for asymmetrical attacks (cf. Section 4).

3.1 Securing the Design

The IK-B Verifier only allows replicas to complete the execution of unmonitored system calls if those calls originate from within an IP-MON instance having a valid one-time authorization token. As only the IK-B Interceptor can generate valid tokens, this mechanism **forces every unmonitored system call to go through IK-B.** At the same time, it also ensures that **IP-MON can only execute unmonitored system calls if it is invoked by IK-B and it is invoked through its intended entry point**. This mechanism is, in essence, a form of Control-Flow Integrity [1]. It also allows us to hide the location of the RB, thereby preventing the RB from being accessed from outside IP-MON. Protecting the RB is of critical importance to the security of our MVEE, as we will discuss in Section 4. To fully hide the location of the RB, while still allowing benign accesses, we ensure that the pointer to the RB is only stored in kernel memory.

IK-B loads the RB pointer and the token into designated processor registers whenever it forwards a call to IP-MON, and IP-MON is designed and implemented such that it does not leak these sensitive values into user space-accessible memory. First, we compile IP-MON using `gcc` and use the `-ffixed-reg` option to remove the RB pointer and authorization token's designated registers from `gcc`'s register allocator. This ensures that the sensitive values never leak to the stack, nor to any other register. Second, we carefully crafted specialized accessor functions to access the RB. These functions may temporarily load the RB pointer into other registers, e.g., to calculate a pointer to a specific element in the RB, but they restore these registers to their former values upon returning. We also force IP-MON to destroy the RB pointer and authorization token registers themselves upon returning to the system call site. Finally, we use inlining to avoid indirect control flow instructions from IP-MON's system call entry point. This ensures that IP-MON's control flow cannot be diverted to a malicious function that could leak the RB pointer or authorization token.

ReMon further prevents discovery of the RB through the /proc/maps interface: It forcibly forwards all system calls accessing the maps file to GHUMVEE and by filtering the data read from the file. This requires marking the maps file as a special file, as described in Section 3.6.

To prevent IP-MON itself from being tampered with, we also force all system calls that could adversely affect IP-MON to be forwarded to GHUMVEE. These calls (e.g. `sys_mprotect` and `sys_mremap`) are then subject to the default lockstep synchronization mechanism.

3.2 The IP-MON Replication Buffer

IP-MON must be embedded into all replicas, so it consists of multiple independent copies, one per replica. These copies must cooperate, which requires an efficient communication channel. Although a socket or FIFO could be used, we opted for a shared replication buffer (RB) stored in a memory segment shared between all replicas.

To increase the scalability of our design, we opted not to use a true circular buffer. Instead, we use a linear RB. When our RB overflows, we signal GHUMVEE using a system call. GHUMVEE then waits for all replicas to synchronize, resets the buffer to its initial state, and resumes the replicas. Involving GHUMVEE as an arbiter avoids costly read-write sharing on RB variables that keep track of where data starts and ends in the RB. Instead, each replica thread only reads and writes its own RB position.

3.3 Adding System Call Support

ReMon currently supports well over 200 system calls. To provide a fast path, IP-MON supports a subset of 67

```
/* read(int fd, void * buf, size_t count) */
MAYBE_CHECKED(read) {
  // check if our current policy allows us to dispatch read
  // calls on this file as unmonitored calls
  return !can_read(ARG1);
}

CALCSIZE(read) {
  // reserve space for 3 register arguments
  COUNTREG(ARG);
  COUNTREG(ARG);
  COUNTREG(ARG);
  // one buffer whose maximum size is in argument 3 of syscall
  COUNTBUFFER(RET, ARG3);
}

PRECALL(read) {
  // compare the args each replica passed to the call.
  // if they match, we allow only the master to complete the call,
  // while the slaves wait for the master's results.
  CHECKREG(ARG1);
  CHECKPOINTER(ARG2);
  CHECKREG(ARG3);
  return MASTERCALL | MAYBE_BLOCKING(ARG1);
}

POSTCALL(read) {
  // replicate the results
  REPLICATEBUFFER(ARG2, ret);
}
```

Listing 1: Replicating the `read` system call in IP-MON.

system calls. However, adding support to IP-MON for a new system call is generally straightforward. IP-MON offers a set of C macros to easily describe how to handle the replication of the system call and its results.

As an example, listing 1 shows IP-MON's code for the `read` system call. The code is split across four handler functions that each implement one step in the handling of a system call using the C macros provided by IP-MON.

First, the MAYBE_CHECKED function is called to determine if the call should be monitored by GHUMVEE. If the MAYBE_CHECKED handler returns `true`, IP-MON forces the original system call to be forwarded to GHUMVEE (④) by destroying the authorization token and restarting the call. We use this handler type to support conditional relaxation policies, as shown in Table 1.

IP-MON uses a fixed-size RB to replicate system call arguments, results, and other system call metadata. Prior to restarting the forwarded call, we therefore need to calculate the maximum size this information may occupy in the RB. If the size of the data as calculated by the CALCSIZE handler exceeds the size of the RB, IP-MON forces the original system call to be forwarded to GHUMVEE. If the data size does not exceed the size of the RB, but it is bigger than the available portion of the RB, the master waits for the slaves to consume the data already in the RB, after which it resets the RB.

Next, if IP-MON has decided not to forward the original system call to GHUMVEE, it calls the PRECALL handler. In the context of the master replica, this function logs the forwarded call's arguments, call number, and a small amount of metadata into the RB. This metadata consists of a set of boolean flags that indicate whether or not the master has forwarded the call to GHUMVEE, whether or not the call is expected to block when it is resumed, etc. If the function is called in a slave replica's context, IP-MON performs sanity checking by comparing the slave's arguments with the master's arguments. If they do not match, IP-MON triggers an intentional crash, thereby signalling GHUMVEE through the `ptrace` mechanism, and causing a shutdown of the MVEE.

The return value of the PRECALL handler determines whether the original call should be resumed or aborted. By returning the `MASTERCALL` constant from the PRECALL handler, for example, IP-MON instructs the master replica to resume the original call, and the slave replicas to abort the original call. Alternatively, the original call may be resumed or aborted in all replicas.

Finally, IP-MON calls the POSTCALL handler. Here, the master replica copies its system call return values into the RB. The slave replicas instead wait for the return values to appear in the RB. Depending on the aforementioned system call metadata, the handler may wait using a spin-wait loop if the system call was not expected to block, or otherwise a specialized condition variable, whose implementation we describe in Section 3.7.

3.4 System Call Monitoring Policies

There are many ways to draw the line between system calls to be monitored by the CP-MVEE and system calls to be handled by IP-MON. We propose two concrete monitoring relaxation policies.

The first option is **spatial exemption**, where certain system calls are either unconditionally handled by IP-MON and not monitored by GHUMVEE, or handled by IP-MON only if their system call arguments meet certain criteria. Table 1 proposes several predefined levels of spatial exemption, which the program developer or administrator can choose from. Selecting a level enables unmonitored system calls for all calls in that level, as well as all preceding levels. This provides a performance-security trade-off, with lower levels in the table having lower overhead but being potentially less secure.

We picked these system calls so we could maintain a high level of security while still preserving the correctness of the replicas' execution and significantly improving our system's performance. System calls that relate to allocation and management of process resources and threads, as well as signal handling, are always monitored by GHUMVEE. This includes syscalls that (i) allocate, manage and close file descriptors (FDs), (ii) map, manage and unmap memory regions, (iii) create, control and kill threads and processes and (iv) all signal handling sys-

Level and description	*Unconditionally allowed calls*	*Conditionally allowed calls depending on*	
		file type	*op type*
BASE_LEVEL Read-only calls that do not operate on file descriptors and do not affect the file system.	gettimeofday, clock_gettime, time, getpid, gettid, getpgrp, getppid, getgid, getegid, getuid, geteuid, getcwd, getpriority, getrusage, times, capget, getitimer, sysinfo, uname, sched_yield, nanosleep		
NONSOCKET_RO_LEVEL Read-only calls on regular files, pipes, and non-socket file descriptors; read-only calls from file system; write calls on process-local variables.	access, faccessat, lseek, stat, lstat, fstat, fstatat, getdents, readlink, readlinkat, getxattr, lgetxattr, fgetxattr, alarm, setitimer, timerfd_gettime, madvise, fadvise64	read, readv, pread64, preadv, select, poll	futex, ioctl, fcntl
NONSOCKET_RW_LEVEL Write calls on regular files, pipes, and other non-socket file descriptors.	sync, syncfd, fsync, fdatasync, timerfd_settime	write, writev, pwrite64, pwritev	
SOCKET_RO_LEVEL Read calls on sockets.	read, readv, pread64, preadv, select, poll, epoll_wait, recfrom, recvmsg, recvmmsg, getsockname, getpeername, getsockopt		
SOCKET_RW_LEVEL Write calls on sockets.	write, writev, pwrite64, pwritev, sendto, sendmsg, sendmmsg, sendfile, epoll_ctl, setsockopt, shutdown		

Table 1: Monitor levels for spatial system call exemption.

tem calls. We distributed all remaining system calls over the aforementioned levels to allow the programmer/administrator to choose the appropriate balance between performance and security.

The second option is **temporal exemption**, where IP-MON probabilistically exempts system calls from the monitoring policy if similar calls were repeatedly approved by the monitor. We observe that many programs, especially those with high system call frequencies, often repeatedly invoke the same sequence of system calls. If a series of system calls is approved by GHUMVEE, then one possible temporal relaxation policy is to stochastically exempt some fraction of the following identical system calls within some time window or range. Note that temporal relaxation policies must be highly unpredictable; deterministic policies (e.g., "Exempt system calls X, Y, Z from monitoring after N approvals within an M millisecond time window") are insecure. In other words, care must be taken to ensure that temporal relaxation does not allow adversaries to coerce the MVEE into a state where potentially dangerous system calls are not monitored.

3.5 IP-MON Initialization

IK-B does not forward any system calls to IP-MON until IP-MON explicitly registers itself through a new system call we added to the kernel. When this call is invoked, the kernel first attempts to report the call to GHUMVEE, which receives the notification and can decide if it wants to allow IP-MON to register.

The registration system call expects three arguments. The first argument is the set of "unmonitored" calls supported by IP-MON. If the IP-MON registration succeeds, IK-B forwards any system call in this set to IP-MON from that point onwards, as we explained earlier. GHUMVEE can modify this set of system calls, or potentially prevent the registration altogether. The second and third arguments are a pointer to the RB and a pointer to the entry point function that should be invoked when IK-B forwards a call to IP-MON.

The RB pointer must be valid and must point to a writable region. IP-MON must therefore set up an RB that it shares with all other replicas. We use the System V IPC facilities to create, initialize, and map the RB [23]. GHUMVEE arbitrates the RB initialization process to ensure that all replicas attach to the same RB.

3.6 The IP-MON File Map

GHUMVEE arbitrates all system calls that create/modify/destroy FDs, incl. sockets. It thus maintains metadata such as the type of each FD (regular/pipe/socket/poll-fd/special). It also tracks which FDs are in non-blocking mode. System calls that operate on non-blocking FDs always return immediately, regardless of whether or not the corresponding operation succeeds.

Replicas can map a read-only copy of this metadata into their address spaces using the same mechanism we use for the RB. We refer to this metadata as the IP-MON file map. We maintain exactly one byte of metadata per FD, resulting in a page-sized file map. For some system calls, IP-MON uses the file map to determine if the call is to be monitored or not as per the monitoring policy.

3.7 Blocking System Calls

Its file map permits IP-MON to predict whether an unmonitored call can block or not. IP-MON handles blocking calls efficiently. If the master replica knows that a call will block, it instructs the slaves to wait on an optimized and highly scalable IP-MON condition variable until the results become available (as opposed to a slower spin-read loop). IP-MON uses the `futex(7)` API to implement wait and wake operations. This allowed us to implement several optimizations.

For each system call invocation, IP-MON allocates a separate structure within the RB. Each individual structure contains a condition variable. Slave replicas must only wait on the condition variable associated with the system call results they are interested in. Using separate condition variables for each system call invocation prevents an unnecessary bottleneck that would arise when using just a single variable, because the slave replicas might progress at different paces. Furthermore, IP-MON tracks whether or not there are replicas waiting for the results of a specific system call invocation. If none are waiting when the master has finished writing its system call results into the buffer, no `FUTEX_WAKE` operation is needed to resume the slaves. IP-MON does not have to

reuse condition variables because a new condition variable is allocated for each system call invocation. Thus, IP-MON does not have to reset condition variables to their initial state after it has used one to signal slave replicas.

3.8 Consistent Signal Delivery

Signals may introduce divergence among a set of executing replicas. MVEEs therefore typically defer the delivery of signals until they can assert that all replicas are in equivalent states, such as when they are all waiting to enter a system call, as discussed in Section 2.2.

The intricacies of the `ptrace` API make the correct implementation of consistent asynchronous signal delivery challenging, and it becomes even more complicated when introducing IP-MON. Because GHUMVEE does not see any system calls that are dispatched as unmonitored calls, it might indefinitely defer the delivery of incoming signals, thus violating the intended behavior of the replicas. GHUMVEE solves this problem via introspection. When a signal is delivered to the master replica, GHUMVEE first sets a *signals pending* flag, which is stored at the beginning of the RB. Next, GHUMVEE checks whether that replica was executing a system call through IP-MON. GHUMVEE does this by checking if the user-space instruction pointer points to a system call instruction inside the IP-MON executable region. If the master replica was executing a blocking system call, GHUMVEE aborts that call. The kernel automatically aborts blocking system calls, but normally restarts them after the signal handler has been invoked. However, GHUMVEE prevents the kernel from restarting the call. Instead, it resumes the master replica at the return site of the call. The master replica then inspects the *signals pending* flag and restarts the call as a monitored call, allowing it to be intercepted by GHUMVEE.

3.9 Support for `epoll (7)`

Linux 2.5.44 introduced the Linux-specific `epoll` API as a high-performance alternative to `select` and `poll`. Applications can use this interface to get notifications for FD events, e.g., when a socket has received new data or when a connection request has arrived. Modern Linux server applications use `epoll` to handle network requests efficiently on multiple threads.

To minimize the performance overhead, IP-MON needs to support the `epoll` family of system calls. This is not straightforward, however. When registering a FD with `epoll` functions, the application can associate an `epoll_event` structure with that FD. This structure may contain a pointer value that the kernel will return when an event on the FD gets triggered. The `epoll_event` structures are challenging to support in MVEEs. Diversified replicas are likely to use different pointer values for the same logical FD. Blindly replicating the results of a `sys_epoll_wait` event would then return the master's, rather than the calling replica's pointer values.

IP-MON solves this problem by maintaining a shadow mapping between FDs and pointers inside `epoll_event`. When a new FD is registered with `epoll`, IP-MON copies the associated pointer value from the `epoll_event` structure to the mapping. When replicating the results of an `epoll` call, IP-MON uses this mapping to store FDs, rather than pointer values in the master replica, and it maps these FDs back onto the associated pointer values in the slave replicas.

4 Security Analysis

Unlike previous MVEEs, ReMon eschews fixed monitoring policies and instead allows security/performance trade-offs to be made on a per-application basis.

With respect to integrity, we already pointed out that a CP MVEE monitor (and its environment) are protected by (i) running it in an isolated process space protected by a hardware-enforced boundary to prevent user-space tampering with the monitor from within the replicas; (ii) by enforcing lockstep, consistent, monitored execution of all system calls in all replicas to prevent malicious impact of a single compromised replica on the monitor; and (iii) diversity among the replicas to increase the likelihood that attacks cause observable divergence, i.e., that they fail to compromise the replicas in consistent ways.

With those three properties in place, it becomes exceedingly hard for an attacker to subvert the monitor and to execute arbitrary system calls. Nevertheless, MVEEs do not protect against attacks that exploit incorrect program logic or leak information though side-channel attacks. This is similar to many other code-reuse mitigations such as software diversity, SFI, and CFI.

In ReMon, monitored system calls are still handled by a CP monitor, so malicious monitored calls are as hard to abuse as they are in existing CP MVEEs. For unmonitored calls, IP-MON relaxes the first two of the above three properties. The master replicas can run ahead of the slaves and the system call consistency checks in the slaves' IP-MON, so an attacker could try to hijack the master's control with a malicious input to execute at least one, and possibly multiple, unmonitored calls without verification by a slave's IP-MON. An attacker could also attempt to locate the RB and feed malicious data to the slaves, in order to stall them or to tamper with their consistency checks. This way, the attacker could increase the window of opportunity to execute unmonitored calls in the master.

As long as the attacker executes unmonitored calls only according to a given relaxation policy, those capabilities by definition pose no significant security threat: unmonitored calls are exactly those calls that are defined by the chosen policy to pose either no security threat at all, or that pose an acceptable security risk. However, an attacker can also try to bypass IP-MON's policy verification checks on conditionally allowed system calls to let IP-MON pass calls unmonitored that should have been

monitored by GHUMVEE according to the policy. We therefore consider several aspects of these attack scenarios in the following paragraphs.

Unmonitored execution of system calls ReMon ensures that IP-MON can only execute unmonitored system calls if it is invoked by IK-B itself and through its intended system call entry point. When invoked properly, IP-MON performs policy verification checks on conditionally allowed system calls, as well as the security checks a CP monitor normally performs. An attacker that manages to compromise a program replica could jump over these checks in an attempt to execute unmonitored system calls directly. Such an attack would, however, be ineffective thanks to the authorization mechanism we described in Section 3.1.

Manipulating the RB We designed IP-MON so that it never stores a pointer to the RB, nor any pointer derived thereof, in user-space accessible memory. Instead, IK-B passes an RB pointer to IP-MON, and IP-MON keeps the RB pointer in a fixed register. To access the RB, the attacker must therefore find its location by random guessing or by mounting side-channel attacks. ReMon's current implementation uses RBs that are 16MiB and located on different addresses in each replica. This gives the RB pointer 24 bits of entropy per replica which makes guessing attacks unlikely to succeed.

Furthermore, because neither IP-MON, nor the application need to know the exact location of the RB and because every invocation of IP-MON is routed through IK-B, we could extend IK-B to periodically move the RB to a different virtual address by modifying the replicas' page table entries. This would further decrease the chances of a successful guessing attack.

Diversified Replicas Our current implementation of ReMon deploys the combined diversification of ASLR and Disjoint Code Layouts (DCL) [40]. ReMon, however, support all other kinds of automated software diversity techniques as well. We refer to the literature for an overview of such techniques [21]. The security evaluations in the literature, including demonstrations of resilience against concrete attacks, therefore still apply to ReMon.

5 Performance Evaluation

In this section, we first evaluate the performance of IP-MON's spatial relaxation policy on a set of widely-used benchmark suites, and then compare IP-MON with existing MVEEs by replicating some of the experiments previously described in the literature [16, 17, 26, 35, 40]. We conducted all of our experiments on a machine with two eight-core Intel Xeon E5-2660 processors each having 20MB of cache, 64GB of RAM and a Gigabit Ethernet connection, running the x86_64 version of Ubuntu 14.04.3 LTS. This machine runs the Linux 3.13.11 kernel, to which we applied the IK-B patches described in Section 3. These IK-B patches add 97 LoC to the kernel. We used the official 2.19 versions of GNU's `glibc` and `libpthreads` in our experiments, but did apply a small patch to `glibc` to reinitialize IP-MON's thread-local storage variables after each fork. We disabled hyper-threading as well as frequency and voltage scaling to maximize reproducibility of our measurements.

Address Space Layout Randomization (ASLR) was enabled in our tests and we configured ReMon to map IP-MON and its associated buffers at non-overlapping addresses in all replicas [40].

5.1 Synthetic Benchmark Suites

We evaluated ReMon on the PARSEC 2.1, SPLASH-2x, and Phoronix benchmark suite. (C. Segulja kindly provided his data race patches for PARSEC and SPLASH [36].) These benchmarks cover a wide range in system call densities and patterns (e.g., bursty vs. spread over time, and mixes of sensitive and non-sensitive calls) as well as various scales and schemes of multi-threading, the most important factors contributing to the overhead of traditional CP-MVEEs that we want to overcome with IP-MON.

We evaluated all five levels of our spatial exemption policy on some of the Phoronix benchmarks, and show the performance of the `NONSOCKET_RW_LEVEL` policy on the other suites. We used the largest available input sets for all benchmarks, and ran the multi-threaded benchmarks with four worker threads and used two replicas for all benchmarks. We excluded PARSEC's `canneal` benchmark from our measurements because it purposely causes data races that result in divergent behavior when running multiple replicas. This makes the benchmark incompatible with MVEEs. We also excluded SPLASH's `cholesky` benchmark due to incompatibilities with the version of the `gcc` compiler we used.

The results for these benchmarks are shown in Figures 3 and 4. The baseline overhead was measured by running ReMon with IP-MON and IK-B disabled. In this configuration, GHUMVEE runs as a standalone MVEE.

GHUMVEE generally performs well in these benchmarks. Our machine can run the replicas on disjoint CPU cores, which means that only the additional pressure on the memory subsystem and the MVEE itself cause performance degradation compared to the benchmarks' native performance. Yet, we still see the effect of enabling IP-MON. For PARSEC 2.1, the relative performance overhead decreases from 21.9% to 11.2%. For SPLASH-2x, the overhead decreases from 29.2% to 10.4%. In Phoronix, the overhead drops from 146.4% to 41.2%. Particularly interesting are the `dedup`, `water_spatial` and `network_loopback` benchmarks, which feature very high system call densities of over 60k system call invocations per second. In these benchmarks, the overheads drop from 252.9% to 69.4%, from 320% to 20.7%, and from

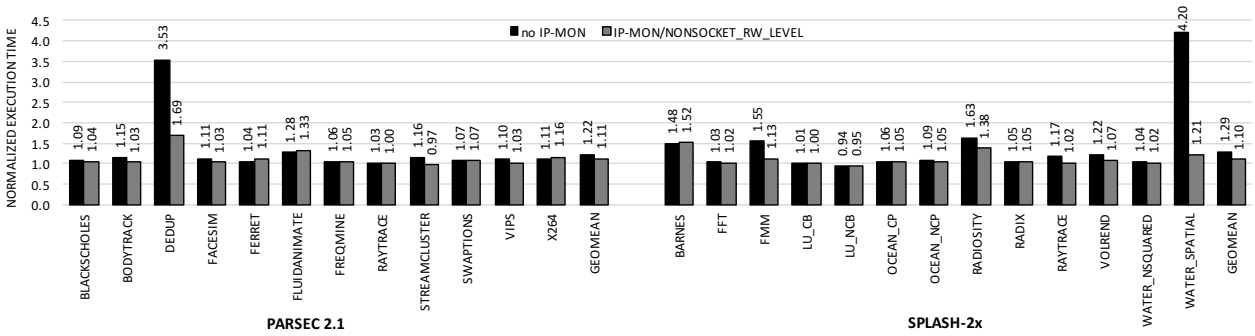


Figure 3: Performance overhead for two benchmark suites (2 replicas).

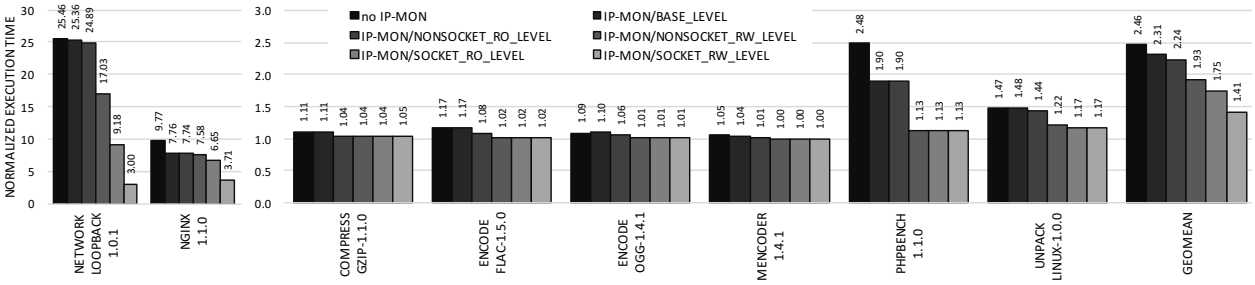


Figure 4: Comparison of IP-MON's spatial relaxation policies in a set of Phoronix benchmarks (2 replicas).

2446% to 200% respectively. Furthermore, the Phoronix results clearly show that different policies allow for different security-performance trade-offs.

5.2 Server Benchmarks

Server applications are great candidates for execution and monitoring by MVEEs because they (i) are frequently targeted by attackers and (ii) often run on many-core machines with idle CPU cores that can run replicas in parallel. In this section, we specifically evaluate our MVEE on applications used to evaluate other MVEEs. These applications include the `Apache` web server (used to evaluate Orchestra [35]), `thttpd (ab)` and `lighttpd (ab)` (used to evaluate Tachyon [26]), `lighttpd (http_load)` (used to evaluate Mx [16]), as well as `beanstalkd`, `lighttpd (wrk)`, `memcached`, `nginx (wrk)` and `redis` (used to evaluate VARAN [17]). We used the same client and server configurations described by the creators of those MVEEs.

We tested IP-MON by running a benchmark client on a separate machine that was connected to our server via a local gigabit link. We evaluated three scenarios. In the first scenario, we used the gigabit link as-is and therefore simulated an unlikely, worst-case scenario since the latency on the gigabit link was very low (less than 0.125ms). In the second scenario, we added a small amount of latency (bringing the total average latency to 2ms) to the gigabit link to simulate a realistic worst-case scenario (average network latencies in the US are 24–63ms [11]). In the third scenario, which we only evaluated to allow for comparison with existing MVEEs, we simulated a total average latency of 5ms. We used Linux' built-in `netem` driver to simulate the latency [24].

Figure 5 shows the worst-case and realistic scenarios side by side. For each benchmark, we measured the overhead IP-MON introduces when running between two and seven parallel replicas with the spatial exemption policy at the `SOCKET_RW_LEVEL`. We also show the overhead for running two replicas with IP-MON disabled. The latter case represents the best-case scenario without IP-MON.

5.3 Comparison with other MVEEs

Table 2 compares ReMon's performance with the results reported for other MVEEs in literature [16, 17, 26, 35, 40] and online [41]. As each MVEE was evaluated in a different experimental setup, the table also lists two features that have a significant impact on the performance overhead. These are the network latencies, because higher latencies hide server-side overhead, as well as the CPU cache sizes, as some of the memory-intensive SPEC benchmarks benefit significantly from larger caches, in particular with multiple concurrent replicas.

From a performance overhead perspective, the worst-case setup in which Mx and Tachyon were evaluated had the benchmark client running on the same (localhost) machine as the benchmark server. For VARAN two separate machines resided in the same rack and were hence connected by a very-low-latency gigabit Ethernet.

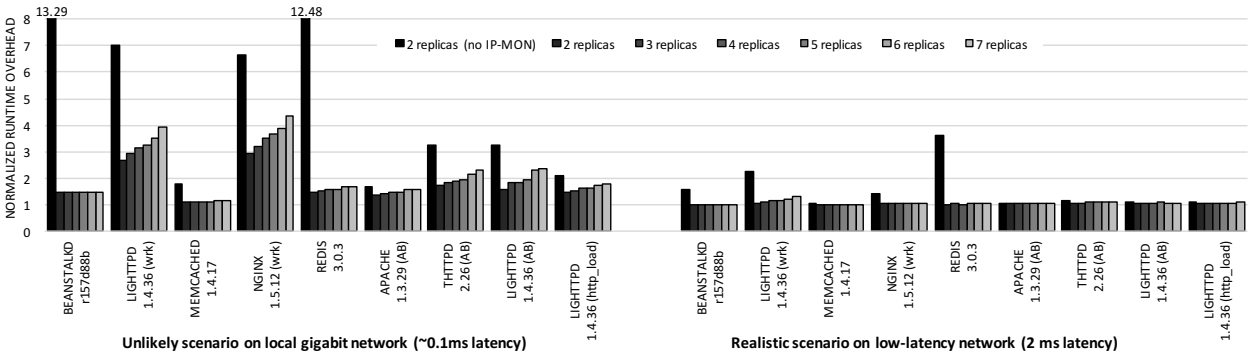


Figure 5: Server benchmarks in two network scenarios for 2–7 replicas with IP-MON and 2 replicas without IP-MON.

Orientation	Reliability						Security			
MVEE	Tachyon			Mx		VARAN	Orchestra	GHUMVEE	ReMon	
network	local-host	local few hops	coast-to-coast	local-host	USA-UK (150 ms)	same rack gigabit	local gigabit	n/a	local gigabit	local gigabit (5 ms)
CPU cache size				8 MB		8 MB		20 MB	20 MB	
reported overheads										
apache (ab)						**2.4%**	50%		34%	**2.4%**
lighttpd (ab)	790%	272%	30%			**0.0%**			55%	**0.0%**
thttpd (ab)	1320%	17%	**0%**			**0.0%**			73%	2.7%
lighttpd (httpld)				249%	4%	**1.0%**			45%	3.5%
redis				1572%	5%	6%			45%	**0.1%**
beanstalkd						52%			45%	**0.6%**
memcached						14%			**8.4%**	**0.3%**
nginx (wrk)						28%			194%	**0.8%**
lighttpd (wrk)						12%			169%	**0.7%**
SPEC CPU2006				17.9%				7.2%	**3.1%**	
SPECint 2006				17.6%		14.2%		12.1%	**3.9%**	
SPECfp 2006				18.3%				3.8%	**2.5%**	

Table 2: Comparison with other MVEEs (2 replicas).

The worst-case setups in which ReMon and Orchestra were evaluated consist of two separate machines connected by a low-latency gigabit link. In these unlikely, worst-case scenarios for servers, the differences in setups hence favor ReMon and Orchestra over VARAN, and VARAN over Tachyon and Mx.

In the best-case setups in which Mx and Tachyon were evaluated, one of the machines was located at the US west coast, while the other was located in England (Mx) or the US east coast (Tachyon). In ReMon's best-case setup, we used a gigabit link with a simulated 5 ms latency. So in the more realistic setups and for the server benchmarks, the differences favor Mx and Tachyon over ReMon.

This comparison demonstrates that ReMon outperforms existing non-hardware assisted security-oriented MVEEs while approaching the efficiency of reliability-oriented MVEEs.

6 Related Work

Directly intercepting system calls—known as **system call interposition**—to check if they are in line with a system call policy (often obtained through profiling and software analysis) predates MVEEs as a security sandboxing technique. The initial literature on the subject identified [15] the high overhead of `ptrace` on Linux (compared to similar techniques on other OSes), and kernel-based implementations were presented to overcome this overhead [31]. To reduce the impact on the kernel, ReMon performs most monitoring in-process, and requires only a small kernel patch to ensure its security.

Dune provides **in-process but across-privilege-ring monitoring** capabilities based on modern x86 hardware virtualization support such as VT-x and Extended Page Tables (EPT) [5]. Dune is, however, currently not thread-safe. This limits its practical applicability.

Cox et al. presented and evaluated an **IP kernel-space MVEE** implementation that deployed address-space partitioning as a diversification technique [12], which can be seen as a limited form of DCL [40]. They measured Apache latency increases of 18% on unsaturated servers, and throughput decreases of 48% on saturated servers, which exceed the corresponding overheads for ReMon.

Later **CP user-space** MVEEs, including the one by Bruschi et al. [8], Orchestra by Salamat et al. [35], and GHUMVEE [42] rely on, and suffer from, the properties of the `ptrace` and `waitpid` APIs. These MVEEs mainly differ from ReMon in the way they perform I/O replication. The Orchestra monitor executes I/O operations on behalf of the replicas, whereas most other MVEEs allow a designated master replica to execute I/O operations. Orchestra copies the results of I/O system calls to the replicas through a shared memory buffer, while Bruschi et al.'s MVEE uses `ptrace` to copy results. GHUMVEE initially relied on a custom `ptrace` implementation to copy data, but now uses the `process_vm_readv` API that was introduced in Linux 3.2.

VARAN takes this approach one step further, and also performs **IP user-space monitoring** [17] through shared ring buffers as shown in Figure 1(b) to avoid the overhead of `ptrace`. In VARAN, the direct master-slave communication is implemented by rewriting the system call instructions (incl. VDSO ones) in the binaries into trampolines to system call replication agents. The agents in the master replica execute the I/O system calls and

log them in the shared buffer. The agents in the slave replicas running behind the master then copy the results instead of executing the calls. Monitors embedded in replica processes check the system call consistency, and can even allow small discrepancies between the system calls behavior of the replicas. VARAN does not replicate user-space synchronization events, however, and hence cannot handle many typical client-side applications, most of which rely on user-space futexes.

With its support for small system call behavior discrepancies, as well as with some of its design and implementation options to minimize overhead, VARAN positions itself as a reliability-oriented MVEE that can support applications such as transparent failover, multi-revision execution (possibly to detect attacks, but not to prevent them), live sanitization, and record-replay [17]. With its in-process replication avoiding `ptrace`, VARAN significantly outperforms Tachyon [26] and Mx [16], two other reliability-oriented MVEEs.

As already noted by its authors, however, VARAN is less fit to protect against memory exploits. First, VARAN lets the master run ahead of the slaves, even for sensitive system calls, as it does not differentiate between sensitive and insensitive calls. This leaves a much larger window of opportunity to attackers than ReMon, including for the execution of sensitive calls. Although this window can be shortened by decreasing the size of VARAN's shared ring buffer, it is unclear what the impact on performance would be and whether that buffer adaptation closes the window completely or merely shortens it to one sensitive system call, which would clearly still be too much. Second, unlike the many protection techniques implemented for ReMon's IP-MON, VARAN's IP monitors are only protected from code-reuse attacks by ASLR, which has proven susceptible to attacks due to low entropy and granularity [3, 18, 37, 38]. This is all the more problematic as VARAN's IP monitors also monitor sensitive system calls. Finally, VARAN only rewrites explicit system call instructions in binary code into trampolines to its replication agents. ReMon, by contrast, intercepts all executed system calls, including any potential unaligned system call gadgets, which would not be identified by VARAN.

MvArmor leverages Dune's aforementioned hardware-assisted monitoring capabilties to offer secure in-process monitoring [20]. MvArmor's performance results are comparable to ReMon's, but due to limitations in Dune, it currently does not support multi-threaded replicas.

SFI [19, 27, 43, 44] and CFI [2, 1, 9] are two defenses that have received a lot of attention in literature which MVEEs can use to protect against memory exploits. Compared to MVEEs such as ReMon, they have the drawback of depending on relatively intrusive code transformations, most of which can only be applied when source code is available, and most of which, in particular those with stronger security guarantees, come with a significant performance penalty.

7 Conclusions

Designers of MVEEs face the mutually conflicting goals of security and runtime performance. Specifically, frequent interactions between cross-process MVEE monitors and program replicas require a high number of costly context switches. We demonstrate a best-of-both-worlds design, ReMon, in which an in-process monitor replicates inputs among the replicas and a cross-process monitor enforces lockstep execution of potentially harmful system calls; innocuous system calls, on the other hand, proceed without external monitoring to increase efficiency.

We present a careful and detailed security analysis and conclude that our introduction of an IP-MON component and relaxed monitoring of innocuous system calls still offers a level of security comparable to that of cross-process MVEEs. Our extensive performance evaluation shows that the overheads of ReMon ranges from 0-3.5% on realistic server workloads and compares very favorably to recent in-process MVEE designs.

Acknowledgments

The authors thank Brian Belleville, Haibo Chen, our reviewers, the Agency for Innovation by Science and Technology in Flanders (IWT), and the Fund for Scientific Research - Flanders.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award number CNS-1513837 as well as gifts from Mozilla, Oracle, and Qualcomm.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, or any other agency of the U.S. Government.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, ÚLFAR., AND LIGATTI, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (2005), ACM, pp. 340–353.

[2] ABADI, M., BUDIU, M., ERLINGSSON, ÚLFAR., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC) 13* (2009), 4:1–4:40.

[3] BARRESI, A., RAZAVI, K., PAYER, M., AND GROSS, T. R. CAIN: Silently breaking ASLR in the cloud. In *USENIX Workshop on Offensive Technologies (WOOT)* (2015), WOOT'15.

[4] BASILE, C., KALBARCZYK, Z., AND IYER, R. A preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks (DSN'02)* (2002), pp. 149–158.

[5] BELAY, A., BITTAU, A., MASHTIZADEH, A., TEREI, D., MAZIÈRES, D., AND KOZYRAKIS, C. Dune: Safe user-level access to privileged CPU features. In *USENIX Symposium on Operating Systems Design and Implementation* (2012), OSDI '12, pp. 335–348.

[6] BERGAN, T., ANDERSON, O., DEVIETTI, J., CEZE, L., AND GROSSMAN, D. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *ACM SIGARCH Computer Architecture News 38*, 1 (2010), 53–64.

[7] BERGER, E. D., AND ZORN, B. G. DieHard: probabilistic memory safety for unsafe languages. In *ACM SIGPLAN Notices* (2006), vol. 41, ACM, pp. 158–168.

[8] BRUSCHI, D., CAVALLARO, L., AND LANZI, A. Diversified process replicae for defeating memory error exploits. In *IEEE International Performance Computing and Communications Conference* (2007).

[9] BUDIU, M., ERLINGSSON, U., AND ABADI, M. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability* (2006), ASID '06.

[10] CAVALLARO, L. *Comprehensive Memory Error Protection via Diversity and Taint-Tracking*. PhD thesis, PhD dissertation, Universita Degli Studi Di Milano, 2007.

[11] COMMISSION, F. C. Measuring broadband America - 2014. `https://www.fcc.gov/reports/measuring-broadband-america-2014`, 2014.

[12] COX, B., EVANS, D., FILIPI, A., ROWANHILL, J., HU, W., DAVIDSON, J., KNIGHT, J., NGUYEN-TUONG, A., AND HISER, J. N-variant systems: a secretless framework for security through diversity. In *USENIX Security Symposium* (2006), USENIX Association, p. 9.

[13] DEVIETTI, J., LUCIA, B., CEZE, L., AND OSKIN, M. DMP: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 85–96.

[14] GARFINKEL, T., PFAFF, B., ROSENBLUM, M., ET AL. Ostia: A delegating architecture for secure system call interposition. In *NDSS '04* (2004).

[15] GOLDBERG, I., WAGNER, D., THOMAS, R., BREWER, E. A., ET AL. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography* (1996), vol. 6.

[16] HOSEK, P., AND CADAR, C. Safe software updates via multi-version execution. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 612–621.

[17] HOSEK, P., AND CADAR, C. VARAN the Unbelievable: An efficient n-version execution framework. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015), ACM, pp. 339–353.

[18] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy* (2013), S&P'13, pp. 191–205.

[19] HUNT, G. C., AND LARUS, J. R. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review 41*, 2 (2007), 37–49.

[20] KONING, K., BOS, H., AND GIUFFRIDA, C. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2016).

[21] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. SoK: Automated software diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2014), S&P '14.

[22] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Respec: efficient online multiprocessor replayvia speculation and external determinism. *ACM SIGARCH Computer Architecture News 38*, 1 (2010), 77–90.

[23] MAN-PAGES PROJECT, T. L. shmop(2) - linux manual page. `http://man7.org/linux/man-pages/man2/shmat.2.html`.

[24] MAN-PAGES PROJECT, T. L. tc-netem(8) - linux manual page. `http://man7.org/linux/man-pages/man8/tc-netem.8.html`.

[25] MANUAL PAGES, O. pledge - restrict system operations. `http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man2/pledge.2`.

[26] MAURER, M., AND BRUMLEY, D. Tachyon: Tandem execution for efficient live patch testing. In *USENIX Security Symposium* (2012), pp. 617–630.

[27] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Usenix Security* (2006), p. 15.

[28] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), PLDI '09.

[29] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M., AND ZDANCEWIC, S. CETS: Compiler enforced temporal safety for C. In *International Symposium on Memory Management* (2010), ISMM '10.

[30] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices 44*, 3 (2009), 97–108.

[31] PROVOS, N. Improving host security with system call policies. In *USENIX Security Symposium* (2002).

[32] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (Berkeley, CA, USA, 2003), SSYM'03, USENIX Association, pp. 18–18.

[33] RONSSE, M., AND DE BOSSCHERE, K. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems (TOCS) 17*, 2 (1999), 133–152.

[34] RUSSINOVICH, M., AND COGSWELL, B. Replay for concurrent non-deterministic shared-memory applications. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'96)* (1996), ACM.

[35] SALAMAT, B., JACKSON, T., GAL, A., AND FRANZ, M. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems* (2009), EuroSys'09, ACM, pp. 33–46.

[36] SEGULJA, C., AND ABDELRAHMAN, T. S. What is the cost of weak determinism? In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (2014), ACM, pp. 99–112.

[37] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security* (2004), CCS '04.

[38] SIEBERT, J., OKHRAVI, H., AND SÖDERSTRÖM, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security* (2014), CCS '14.

[39] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *Proceedings of the 35th IEEE Symposium on Security and Privacy* (2013), S&P '13.

[40] VOLCKAERT, S., COPPENS, B., AND DE SUTTER, B. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing PP*, 99 (2015).

[41] VOLCKAERT, S., AND DE SUTTER, B. GHUMVEE website. `http://ghumvee.elis.ugent.be`.

[42] VOLCKAERT, S., DE SUTTER, B., DE BAETS, T., AND DE BOSSCHERE, K. GHUMVEE: efficient, effective, and flexible replication. In *5th International Symposium on Foundations and practice of security (FPS 2012)* (2013), Springer, pp. 261–277.

[43] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review* (1994), vol. 27, ACM, pp. 203–216.

[44] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy* (2009), IEEE, pp. 79–93.

Blockstack: A Global Naming and Storage System Secured by Blockchains

Muneeb Ali[⋆†], Jude Nelson[⋆†], Ryan Shea[†], Michael J. Freedman[⋆]
[⋆]*Princeton University,* [†]*Blockstack Labs*

Abstract

Blockchains like Bitcoin and Namecoin and their respective P2P networks have seen significant adoption in the past few years and show promise as naming systems with no trusted parties. Users can register human meaningful names and securely associate data with them, and only the owner of the particular private keys that registered them can write or update the name-value pair. In theory, many decentralized systems can be built using these blockchain networks, such as new, decentralized versions of DNS and PKI. As the technology is relatively new and evolving rapidly, however, little production data or experience is available to guide design tradeoffs.

In this paper, we describe our experiences operating a large deployment of a decentralized PKI service built on top of the Namecoin blockchain. We present various challenges pertaining to network reliability, throughput, and security that we needed to overcome while registering and updating over 33,000 entries and 200,000 transactions on the Namecoin blockchain. Further, we discuss how our experience informed the design of a new blockchain-based naming and storage system called Blockstack. We detail why we switched from the Namecoin network to the Bitcoin network for the new system, and present operational lessons from this migration. Blockstack is released as open source software and currently powers a production PKI system for 55,000 users.

1 Introduction

Cryptocurrency blockchains and their respective P2P networks are useful beyond exchanging money. They provide cryptographically auditable, append-only ledgers that are already being used to build new, *decentralized* versions of DNS [41] and public-key infrastructure (PKI) [43], along with other applications like file storage [23] and document timestamping [15]. Because blockchains have no central points of trust or failure, they enable a new class of decentralized applications and services that minimize the degree to which users need to put trust in a single party, like a DNS root server or a root certificate authority.

Blockchain networks have attracted a lot of interest from enthusiasts, engineers, and investors. In fact, 1.1 billion USD has been invested in blockchain startups over the last several years [19]. With the rapid capital infusion, infrastructure for blockchains is getting quickly deployed [18] and blockchains are emerging as publicly available common infrastructure for building decentralized systems and applications. However, blockchain networks are at a very early stage and there is very little production data available to guide design trade-offs.

Many non-financial applications of blockchains imply the need for a naming system that securely binds *names*, which can be human-readable, to arbitrary *values*. The blockchain gives consensus on the global state of the naming system and provides an append-only global log for state changes. Writes to name-value pairs can only be announced in new blocks, as appends to the global log. The global log is logically centralized (all nodes on the network see the same state), but organizationally decentralized (no central party controls the log).

The decentralized nature of blockchain-based naming introduces meaningful security benefits, but certain aspects of contemporary blockchains present technical limitations. Individual blockchain records are typically on the order of kilobytes [49] and cannot hold much data. Latency of creating and updating records is capped by the blockchain's write propagation and leader election protocol, and it is typically on the order of 10-40 minutes [14]. The total new operations in each round are limited by average bandwidth of nodes participating in the network (for Bitcoin the current average is $\sim$1500 new operations per new round [2]). Further, new nodes need to independently audit the global log from the beginning: as the system makes forward progress, the time to bootstrap new nodes increases linearly.

We believe that in spite of these scalability and performance challenges, blockchains provide important infrastructure for building secure, decentralized services. The cost of tampering with blockchains grows with their adoption: today, it would require hundreds of millions of dollars to attack a large blockchain like Bitcoin [1].

These benefits motivated us to use blockchains to build a new decentralized PKI system. Our system enables users to register unique, human-readable usernames and associate public-keys, like PGP [53], along with additional data to these usernames. There is no need for any central or trusted party in our PKI system. This paper presents our experiences from operating this PKI system on the Namecoin network, which is one of the largest services built on top of a blockchain to date. We outline the challenges that we had to overcome for registering and updating over 33,000 user entries and for sending over 200,000 transactions on the Namecoin network.

Our production deployment led to many interesting experiences where we observed and analyzed network anomalies and security problems that were not discovered or documented before. **We discovered a critical security problem where a single miner consistently had more than 51% of the total compute power on the Namecoin network** (see [35] for details on the 51% attack and compute power of miners). A 51% attack is one of the most serious attacks on a blockchain and impacts its security and decentralization properties.

Moreover, we also encountered chronic networking issues with broadcasting transactions on the Namecoin network. Reliability of the network generally depends on how actively a blockchain network is monitored and maintained, as well as the financial incentives for operating the network. Therefore, for both security and reliability reasons, blockchain-based services should use the largest and most secure blockchain, which at the time of writing is the Bitcoin blockchain.

Our experience with Namecoin informed the design and implementation of a new blockchain-based naming and storage system, called Blockstack, that uses the Bitcoin blockchain. Unlike previous blockchain-based systems, Blockstack *separates its control and data plane considerations*: it keeps only minimal metadata (namely, data hashes and state transitions) in the blockchain and uses external datastores for actual bulk storage. Blockstack enables fast bootstrapping of new nodes by using checkpointing and *skip lists* to limit the set of blocks that a new node must audit to get started. We have released Blockstack as open source [13].

Modifying production decentralized systems like Bitcoin (and introducing new functionality for which it was not designed) is quite difficult, particularly that the system still needs to reach "consensus." With Blockstack, we extend the single state machine model of blockchains to allow for arbitrary state machines without requiring consensus breaking changes in the underlying blockchain. This design was non-intuitive before our work; indeed, the standard approach for the past three years was to fork the main Bitcoin blockchain to add new and different functionality. Our experience with the Namecoin blockchain shows that starting new, smaller blockchains leads to security problems (like reduced computational power needed to attack the network) and should be avoided when possible.

This paper makes the following contributions:

- We present the first analysis of security and network reliability of a blockchain other than Bitcoin and report a critical security problem where a major alternate blockchain, Namecoin, had a single miner with well over 51% of the compute power for months.
- We report that merged mining, a popular method to secure smaller blockchains, is currently failing in practice. The total compute power dedicated to blockchains is currently insufficient to support multiple secure blockchains.
- We present the design of Blockstack's logically separate layer, *virtualchain*, which introduces novel new functionality to production blockchains without requiring any consensus-breaking changes from the underlying blockchain.
- We present a migration framework for migrating from one blockchain to another under a failure of the underlying blockchain, and present lessons from a successful migration of our production system from Namecoin to Bitcoin. This was the first cross-chain migration of a production system running on blockchains.

2 Motivation and Background

In this section, we describe the motivation for building naming systems that have no central point of trust and provide the relevant background on blockchains. In this paper, we use the term *naming system* to mean (a) names are **human-readable** and can be picked by humans, (b) name-value pairs have **a strong sense of ownership**—that is, they can be owned by cryptographic keypairs, and c) there is **no central trusted party** or point of failure. Building a naming system with these three properties was considered impossible according to Zooko's Triangle [32] and most traditional naming systems provide two out of these three properties [31]. Namecoin [41] used a blockchain-based approach to provide the first naming system that offered all three properties: human-readability, strong ownership, and decentralization.

2.1 Background on Blockchains

Blockchains provide a global append-only log that is publicly writeable. Writes to the global log, called *transactions*, are organized as *blocks* and each block packages multiple transactions into a single atomic write. Writing to the global log requires a payment in the form of a *transaction fee*. Nodes participating in a blockchain network follow a leader election protocol for deciding which node gets to write the next block and collect the respective transaction fees. Not all nodes in the network participate in leader election. Nodes actively competing to become the leader of the next round are called *miners*. At the start of each round, all miners start working on a new computation problem, derived from the last block, and the miner that is the first to solve the problem gets to write the next block. In Bitcoin, the difficulty of these computation problems is automatically adjusted by the protocol so that 1 new block is produced roughly every 10 minutes. See [14] for further details on how blockchains work and how they reach consensus.

2.2 Namecoin's Naming System

Namecoin is one of the first forks of Bitcoin and is the oldest blockchain other than Bitcoin that is still operational, with a cryptocurrency market capitalization of 5 million USD as of May 2016 [6] (the market capitalization of a cryptocurrency is the exchange-traded value of its coins multiplied by its number of coins in existence). The main motivation for starting Namecoin was to create an alternate DNS-like system that replaces DNS root servers with a blockchain for mapping domain names to DNS records [41]. Given that blockchains don't have central points of trust, a blockchain-based DNS is much harder to censor and registered names cannot be seized from owners without getting access to their respective private keys [31]. Altering name registrations stored in a blockchain requires prohibitively high computing resources because re-writing blockchain data requires proof-of-work [8]. Before our work, it was common practice to start new blockchains (by forking them from Bitcoin) to introduce new functionality and make modifications required by the respective service/application, which is the precise approach taken by Namecoin.

Just like DNS, there is a cost associated with registering a new name. The name registration fee discourages people from registering a lot of names that they don't actually intend to use. In Namecoin, the recipient of registration fees is a "black hole" cryptographic address from which money cannot be retrieved [31]. Namecoin defines a pricing function for how the cost of name registrations changes over time. Namecoin supports multiple namespaces (like TLDs in DNS), and the same rules for pricing and name expiration apply to all namespaces. By convention, the $d/$ namespace is used for domain names.

In Namecoin, name registration uses a two-phase commit method where a user first **pre-orders** a name hash and then **registers** the name-value pair by revealing the actual *name* and the associated *value*. This is done to avoid front-running of unconfirmed name registrations [31]. Name registrations expire after a fixed amount of time, measured in new blocks written (currently 36,000 blocks, which translates to roughly 8 months). Namecoin also supports updating the value associated with a name, as well as ownership transfers.

2.3 Blockchain-based PKI System

We used Namecoin to build a PKI and identity system, called *Blockstack ID*, by starting a new namespace $u/$ on it. We defined the format for publishing public keys, like PGP [53], along with other profile data in the blockchain [3]. This is similar to defining the format of DNS records. Namecoin already had support for human-readable names and registering name-value pairs. Namecoin provided limited storage per name-value pair and we extended the storage capacity by using linked lists of name-value pairs. We also improved the read performance of Namecoin for our production system.

We launched a web service [43] in March 2014 that enabled people to easily register names on the $u/$ namespace of Namecoin and associate profile data with them. In our web service, we first register the name on the user's behalf (and also pay the registration fee) and then transfer the name to a cryptocurrency address owned by the user. Our implementation is one of the first production PKI systems that binds user identities to public keys using a blockchain (see Section 6 for other systems). All registered names have an ECDSA public key [28] binding by default, and a subset of users have added their PGP keys as well. According to a study by Harry et al. [31], our system has the second largest namespace on Namecoin by volume and the largest by number of active users.

3 Lessons from Namecoin Deployment

In this section, we describe our experience with running a year-long production system on Namecoin and the challenges we faced. We present lessons we learned for securing blockchains (§3.1, §3.3, and §3.5), improving network reliability (§3.2), and for deploying consensus breaking changing (§3.4). These lessons directly influenced the design of our new system, Blockstack (§4).

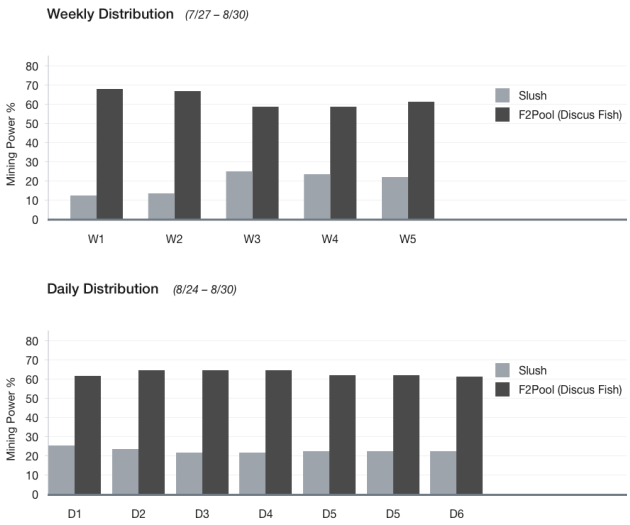


Figure 1: Weekly and daily mining distribution.

3.1 Blockchain Security

The security of name ownership is tied to the security of both the underlying blockchain and the software powering it. The most important factor in the security of a blockchain is the total cost of attacking the blockchain and tampering with recently written data. Miners often pool their resources to form a *mining pool*, which is essentially a super node on the network (a lot of computational power behind a single miner node). If the amount of computational power under the control of a single miner (or pool) is more than the rest of the network, called a *51% attack*, then that miner has the ability to attack the network and rewrite recent blockchain history, censor transactions (e.g., for name registrations), and steal cryptocurrency using *double spend* attacks [49]. This is because it will win the leader election for a majority of the time, and produce a blockchain history with more proof-of-work than any disagreeing miner. The more expensive it is to control a majority of the compute power on a blockchain, the more secure the blockchain.

We noticed in late 2014 that a single mining pool consistently had more than 51% of the compute power on Namecoin. Recently, the situation has been even worse, with a single mining pool controlling over 60% of Namecoin's compute power. Figure 1 shows the weekly and daily distribution of mining power for the month of August 2015, right before we migrated our system away from Namecoin. In fact, we have observed F2Pool (also known as Discus Fish) control up to 75% of compute power in a particular week. At such concentration, Namecoin is effectively controlled by a single party; F2Pool gets to write most of the new blocks and can undermine the security of the blockchain at will.

Other than raw hashing power, software bugs can also introduce security problems, e.g., a Namecoin bug allowed people to steal names from anyone [26]. Denial-of-service attacks are another attack vector; the more peers a cryptocurrency network has, the more resilient the network is to denial-of-service attacks.

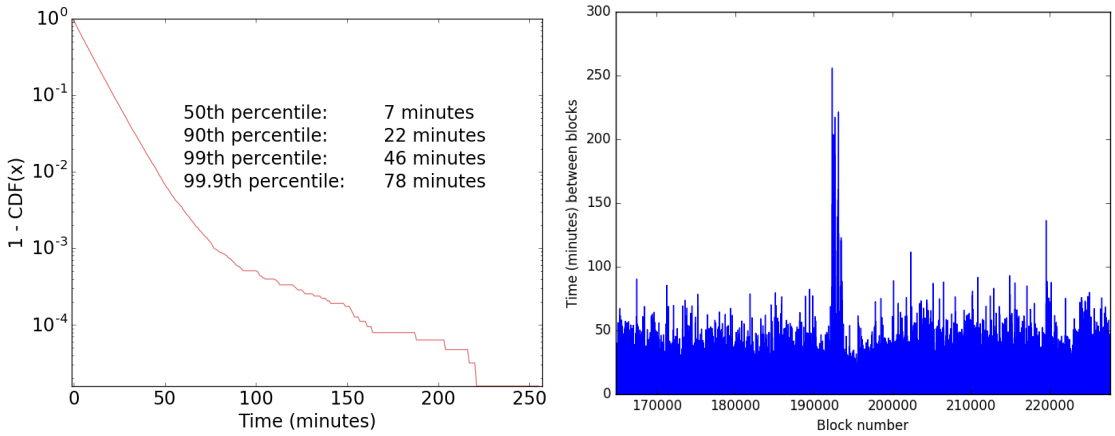
Bitcoin currently has the largest amount of computational power securing the blockchain data. Bitcoin's codebase is more actively developed with more bug bounties than other blockchains. Namecoin has many fewer peer nodes than Bitcoin (170 vs. 4,600 in Jan 2016 [4]), which makes it more vulnerable to DDoS attacks as well. The Bitcoin blockchain is currently by far the most secure blockchain. However, it's extremely hard to introduce new functionality to Bitcoin because that requires consensus-breaking changes (Section 3.4).

Lesson #1: There is a fundamental tradeoff between blockchain security and introducing new functionality to blockchains. Starting a new blockchain network is how developers typically introduce new functionality not provided by Bitcoin, e.g., a naming system that is of interest to many emerging applications. However, new blockchains are significantly less secure than Bitcoin. In Section 4, we introduce Blockstack to overcome this tradeoff by creating *virtualchains* that introduce new functionality as a layer on top of Bitcoin.

3.2 Network Reliability and Throughput

The throughput of our PKI system (number of entries we can register/update) is directly dependent on the throughput of the underlying blockchain. The number of new register/update operations that can be performed per hour is limited by the number of transactions that can be sent (and confirmed) on the underlying blockchain per hour. Similarly, reliability of our PKI system is impacted if the underlying blockchain cannot perform operations reliably and consistently.

Network Latency Spike: As a fork of Bitcoin, Namecoin shares many protocol properties with Bitcoin, including a 10 minute average leader election time (the "latency target") and a 1MB bandwidth limit on block size (giving throughput of $\sim$1000 transactions per block). Figure 2(a) shows that since we launched our PKI system in March 2014, Namecoin on average performed well on the network latency target. As expected, most new blocks were written within 10 and 40 minutes (similar times have also been observed on Bitcoin [14]). Figure 2(b) shows an incident in late August 2014 (at block number 192000), where network latency skyrocketed for a couple of weeks ($\sim$1000 blocks are roughly a week). After investigating the issue and having discussions with Namecoin developers, we discovered that the latency spike was caused by software issues in Namecoin. Someone on the network was sending transactions with a large



(a) CCDF of network latency (03/14 – 04/15)

(b) Network latency per new block (03/14 – 04/15)

Figure 2: Network latency spike in the Namecoin network.

number of data fields per transaction. This was causing severe performance problems for the miners and their Namecoin daemons kept crashing. Without stable miner nodes, blocks were not getting appended in a timely fashion. This shows that unexpected protocol/software issues can trigger network latency problems. During this period, we noticed a slow down in rate of new registrations of our PKI system along with a spike in user complaints.

Network Throughput Drop: In early September 2014, right after the latency spike incident we noticed that our transactions were not getting accepted for many consecutive blocks and, after a while, will get accepted in bulk in a single block that packaged a lot of transactions. We noticed that a lot of new blocks had no transactions in them. This issue persisted for over a week and Figure 3 plots the number of transactions that we were trying to send (shown as "tx target") vs. the number of transactions that were getting accepted by the network. Network latency was completely normal (shown at top of Figure 3), but network throughput went down because of no transactions in new blocks. We tried upgrading our software and rebroadcasting transactions, but the issue persisted. We concluded that there is a large mining pool that is either intentionally refusing or is unable to package transactions in the blocks it is writing. Our transactions will get packaged only when some other miner was elected to write the new block. We discuss this issue in more detail in the next section.

Lesson #2: There is currently a significant difference between the network reliability of the largest public blockchain network (Bitcoin) and network reliability of the long tail of alternate blockchains. Problems with the Bitcoin network impact a lot more users and businesses than Namecoin and other smaller blockchains. Our work is the first analysis of the network reliability of a blockchain other than Bitcoin.

3.3 Potential Selfish Mining

The signs that we noticed in the incident where miners were not accepting our transactions (Section 3.2) looked similar to a selfish mining attack [22]. In a selfish mining attack, (a) a miner needs to have a large amount of mining power (more than 33%), (b) people would notice long delay in blocks followed by blocks in very quick succession, and (c) there will be a lot of rejected blocks. We noticed all these signs, and believe that the unusually high computing power of a single miner led to conditions similar to selfish mining. That is, the miner was able to work on new blocks faster than the others and append them in rapid succession.

Lesson #3: Selfish-mining is not just a theoretical attack, but selfish-mining like behavior can already be observed in production blockchains. This is the first time that data collected from a production network shows signs of selfish-mining like behavior, regardless of if the miner was intentionally attacking the network or not.

3.4 Consensus-breaking Changes

For major updates, like changes to name pricing, Namecoin requires a "hard fork" in which everyone on the network must upgrade their software, and nodes on previous versions can no longer participate in the network. Anecdotal evidence suggests that it's hard to get miners to upgrade their software because they don't have enough incentive to spend engineering hours on maintaining a small cryptocurrency like Namecoin, which is not their main reason for operating a mining pool. Our experience monitoring the Namecoin network showed that whenever

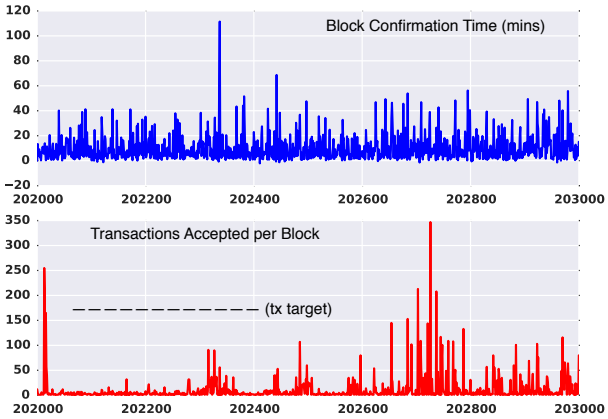


Figure 3: *Throughput drop in the Namecoin network. The number of transactions we were trying to send is shown as "tx target".*

software updates were issued on Namecoin, there was a considerable fluctuation of computing power. In fact, we noticed that after the recent upgrade to Namecoin Core [42], a major upgrade to the Namecoin daemon, many miners dropped out and never came back online.

Lesson #4: Other than the engineering problems, consensus-breaking changes are complicated because of fundamental incentive structures of the parties involved. System designers have never dealt with consensus-breaking changes before cryptocurrencies; it's a novel challenge. For software upgrades to cryptocurrency networks we should: (a) separate consensus-breaking upgrades from other upgrades as a software engineering rule (Bitcoin recently started doing this in their codebase [11]) and (b) try to align miner incentives given their cost (engineering time) of software upgrades. This resistance to upgrades is present in Bitcoin as well, but is exaggerated for the long tail of smaller blockchains. In Section 4, we describe how Blockstack accounts for these incentives and introduces new features without requiring miners to upgrade software.

3.5 Failure of Merged Mining

The security of a blockchain depends on the relative compute power of miners and the cost for a single party to employ more computing power than the rest of the network. New, smaller blockchains have a *bootstrapping problem*, however: in the initial days of a new blockchain, it would be relatively easy for a single party to take it over, since the total compute power on the blockchain is not yet large enough to prevent this. To address this problem, Satoshi Nakamoto (author of Bitcoin) introduced "merged mining" [39], where an alternate blockchain can allow Bitcoin miners to participate in the new network without requiring them to spend extra compute cycles. The miners can make extra profits on the new blockchain without adding computational overhead. With a merge-mined cryptocurrency, the security of the blockchain is typically a subset of the "main blockchain," because in practice not all miners of the main blockchain go through the trouble of setting up merged mining.

Namecoin switched to merged mining with Bitcoin to increase security of its blockchain [31]. Namecoin is the oldest and largest merged-mined cryptocurrency and inspired other cryptocurrencies to consider it as well. One of our key findings is that merged mining is currently failing in practice: the leading merged-mined blockchain, Namecoin, is vulnerable to the 51% attack (Section 3.1). Moreover, merged-mining provided a false sense of security. F2Pool controls 30-35% computing power of Bitcoin, but over 60% of Namecoin's computing power through merged mining, leaving Namecoin vulnerable to a 51% attack. Unless the merged mined cryptocurrency can consistently attract a very high ratio of main blockchain miners to support their software, merged mining will not keep it safe from 51% attacks.

Lesson #5: At the current stage in the evolution of blockchains, there are not enough compute cycles dedicated to mining to support multiple secure blockchains. The respective financial capital attached to blockchains relative to Bitcoin supports this argument: as of Feb 2016, Bitcoin has a 5.9 billion USD market cap, which accounts for 89% of the market cap of all 500+ blockchains combined, while the second and third largest market caps are 3.2% and 2.6% of Bitcoin, respectively [6]. While multiple secure blockchains may be possible after the technology matures and enjoys wider adoption, in the near future, Bitcoin's blockchain is the only one that is prohibitively expensive to attack.

3.6 Summary

Namecoin deserves full credit for originally solving naming on a blockchain. But after considering all of the above factors, it was an easy decision to move our PKI system from Namecoin to Bitcoin. In general, after our experience, we strongly believe that decentralized applications and services need to be on the largest, most secure, and most actively maintained blockchain. Currently, no other blockchain even comes close to Bitcoin in terms of these security requirements.

4 Design of Blockstack

Blockstack is designed to implement a naming system with human-readable names in a layer above the blockchain. In this section, we describe how Blockstack

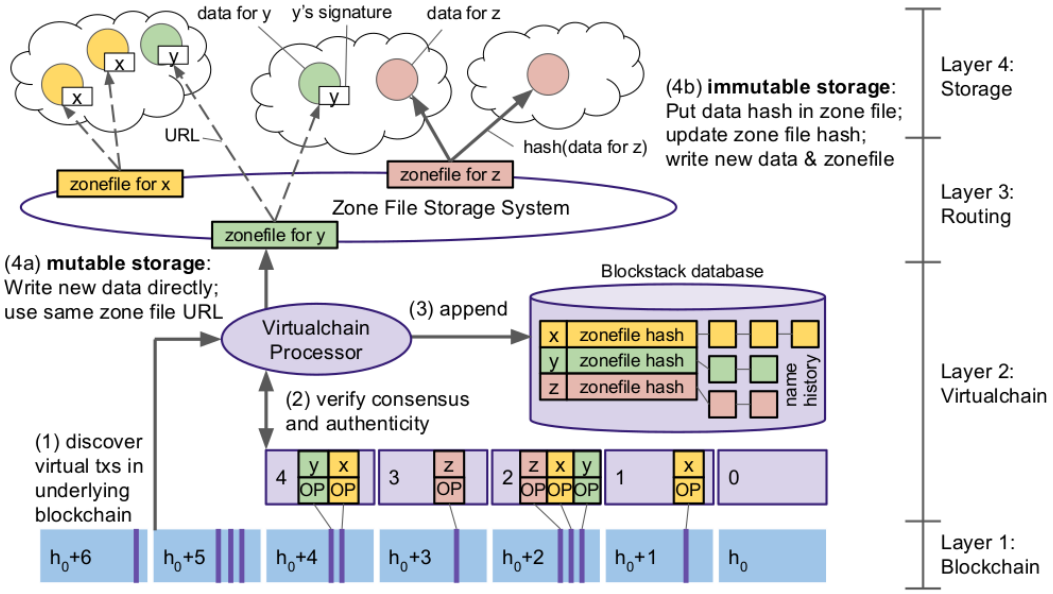


Figure 4: *Overview of Blockstack's architecture. Blockchain records give (name, hash) mappings. Hashes are looked up in routing layer to discover routes to data. Data, signed by name owner's public-key, is stored in cloud storage.*

uses the underlying blockchain, and present how it copes with technical limitations of contemporary blockchains.

4.1 Challenges

Building systems with blockchains presents challenges:

• **Limits on Data Storage:** Individual blockchain records are typically on the order of kilobytes [49] and cannot hold much data. Moreover, the blockchain's log structure implies that *all* state changes are recorded in the blockchain. All nodes participating in the network need to maintain a full copy of the blockchain, limiting the total size of blockchains to what current commodity hardware can support. As of May 2016, Bitcoin nodes need to dedicate 69GB total disk space to blockchain data for staying synchronized with the network.

• **Slow Writes:** The transaction processing rate is capped by the blockchain's write propagation and leader election protocol, and it is pegged to the rate at which new blocks are announced by leader nodes, called *miners* in many blockchain networks [14]. New transactions can take several minutes to a few hours to be accepted.

• **Limited Bandwidth:** The total number of transactions per block is limited by the *block size* of blockchains. To maintain fairness and to give all nodes a chance to become leader in the next round, all nodes should receive a newly announced block at roughly the same time. Therefore, the *block size* is typically limited by average uplink bandwidth of nodes [14]. For Bitcoin the current bandwidth is 1MB (∼1000 transactions) per new block.

• **Endless Ledger:** The integrity of blockchains depends on the ability for anyone to audit them back to the first block. As the system makes forward progress and issues new blocks, the cost of an audit grows linearly with time, which makes booting up new nodes progressively more time consuming. We call this the *endless ledger problem*. Bitcoin's blockchain currently has ∼413,000 blocks and new nodes take 1-3 days to download the blockchain from Bitcoin peers, verify it, and boot up.

4.2 Architecture Overview

Blockstack maintains a naming system as a separate logical layer on top of the underlying blockchain on which it operates. Blockstack uses the underlying blockchain to achieve consensus on the state of this naming system and bind names to data records. Specifically, it uses the underlying blockchain as a communication channel for announcing state changes, as any changes to the state of name-value pairs can only be announced in new blockchain blocks. Relying on the consensus protocol of the underlying blockchain, Blockstack can provide a total ordering for all operations supported by the naming system, like name registrations, updates and transfers.

Separation of the Control and Data Plane: Blockstack decouples the security of name registration and name ownership from the availability of data associated with names by separating the control and data planes.

The control plane defines the protocol for registering human-readable *names*, creating $(name, hash)$ bindings, and creating bindings to owning cryptographic keypairs.

The control plane consists of a blockchain and a logically separate layer on top, called a "virtualchain".

The data plane is responsible for data storage and availability. It consists of (a) zone files for discovering data by hash or URL, and (b) external storage systems for storing data (such as S3, IPFS [29], and Syndicate [30]). Data values are signed by the public keys of the respective name owners. Clients read data values from the data plane and verify their authenticity by checking that either the data's hash is in the zone file, or the data includes a signature with the name owner's public key.

We believe this separation is a significant improvement over Namecoin, which implements both the control and the data plane at the blockchain level. Our design not only significantly increases the data storage capacity of the system, but also allows each layer to evolve and improve independently of the other.

Agnostic of the Underlying Blockchain: The design of Blockstack does not put any limitations on which blockchain can be used with it. Any blockchain can be used, but the security and reliability properties are directly dependent on the underlying blockchain. We believe that the ability to *migrate* from one blockchain to another is an important design choice as it allows for the larger system to survive, even when the underlying blockchain is compromised. Currently, Blockstack core developers decide which underlying blockchain(s) to support in which version of the software. Individual applications can decide to run the software version of their choice and keep their namespace on a particular blockchain, if they prefer not to migrate. Section 5 gives more details on the migration process.

Ability to Construct State Machines: A key contribution of Blockstack is the introduction of a logically separate layer on top of a blockchain that can construct an arbitrary *state machine* after processing information from the underlying blockchain. We call this layer a *virtualchain* (Section 4.3.2). A *virtualchain* treats transactions from the underlying blockchain as inputs to the state machine and valid inputs trigger state changes. At any given time, where time is defined by the block number, the state machine can be in exactly one global state. Time moves forward as new blocks are written in the underlying blockchain and the global state is updated. **A virtualchain can introduce new types of state machines without requiring any changes from the underlying blockchain.** Introducing new state machines directly in a blockchain requires peers to upgrade. Upgrades potentially break consensus and cause forks. In practice, they are difficult to orchestrate [14]. Currently, Blockstack introduces a state machine that represents the global state of a naming system, including who owns a particular name and what data is associated with a name. Further, it's possible to use the *virtualchain* concept to define other types of state machines as well.

4.3 Blockstack Layers

Blockstack introduces new functionality on top of blockchains by defining a set of new operations that are otherwise not supported by the blockchain. Blockstack has four layers, with two layers (blockchain layer and *virtualchain* layer) in the control plane and two layers (routing layer and data storage layer) in the data plane.

4.3.1 Layer 1: Blockchain Layer

The blockchain occupies the lowest tier, and serves two purposes: it stores the sequence of Blockstack operations and it provides consensus on the order in which the operations were written. Blockstack operations are encoded in transactions on the underlying blockchain.

4.3.2 Layer 2: Virtualchain Layer

Above the blockchain is a *virtualchain*, which defines new operations without requiring changes to the underlying blockchain. Only Blockstack nodes are aware of this layer and underlying blockchain nodes are agnostic to it. Blockstack operations are defined in the *virtualchain* layer and are encoded in valid blockchain transactions as additional metadata. Blockchain nodes do see the raw transactions, but the logic to process Blockstack operations only exists at the *virtualchain* level.

The rules for accepting or rejecting Blockstack operations are also defined in the *virtualchain*. Accepted operations are processed by the *virtualchain* to construct a database that stores information on the global state of the system along with state changes at any given blockchain block. Virtualchains can be used to build a variety of state machines. Currently, Blockstack defines only a single state machine - a global naming and storage system.

4.3.3 Layer 3: Routing Layer

Blockstack separates the task of *routing* requests (i.e., how to discover data) from the actual storage of data. This avoids the need for the system to adopt any particular storage service from the onset, and instead allows multiple storage providers to coexist, including both commercial cloud storage and peer-to-peer systems.

Blockstack uses *zone files* for storing routing information, which are identical to DNS zone files in their format. The *virtualchain* binds *names* to respective $hash(zonefile)$ and stores these bindings in the control plane, whereas the *zone files* themselves are stored in the routing layer. **Users do not need to trust the routing layer** because the integrity of *zone files* can be verified by checking the $hash(zonefile)$ in the control plane.

In Blockstack's current implementation, nodes form a DHT-based peer network [36] for storing *zone files*. The DHT only stores *zone files* if $hash(zonefile)$ was previously announced in the blockchain. This effectively whitelists the data that can be stored in the DHT. Due to space constraints, we omit most details of our DHT storage from this paper; the key aspect relevant to the design of Blockstack is that routes (irrespective of where they are fetched from) can be verified and therefore cannot be tampered with. Further, most production servers maintain a full copy of all *zone files* since the size of *zone files* is relatively small (4KB per file). Keeping a full copy of routing data introduces only a marginal storage cost on top of storing the blockchain data.

4.3.4 Layer 4: Storage Layer

The top-most layer is the storage layer, which hosts the actual data values of name-value pairs. All stored data values are signed by the key of the respective owner of a *name*. By storing data values outside of the blockchain, Blockstack allows values of arbitrary size and allows for a variety of storage backends. **Users do not need to trust the storage layer** because they can verify the integrity of the data values in the control plane.

There are two modes of using the storage layer and they differ in how the integrity of data values is verified; Blockstack supports both storage modes simultaneously.

(a) Mutable Storage is the default mode of operation for the storage layer. The user's zone file contains a URI record that points to the data, and the data is constructed to include a signature from the user's private key. Writing the data involves signing and replicating the data (but not the zone file), and reading the data involves fetching the zone file and data, verifying that $hash(zonefile)$ matches the hash in Blockstack, and verifying the data's signature with the user's public key. This allows for writes to be as fast as the signature algorithm and underlying storage system allows, since updating the data does not alter the zone file and thus does not require any blockchain transactions. However, readers and writers must employ a data versioning scheme to avoid consuming stale data.

(b) Immutable Storage is similar to mutable storage, but additionally puts a TXT record in the zone file that contains $hash(data)$. Readers verify data integrity by fetching the data and checking that $hash(data)$ is in the zone file, in addition to verifying the data's signature and the zone file's authenticity. This mode is suitable for data values that don't change often and where it's important to verify that readers see the latest version of the data value. For immutable storage, updates to data values require a new transaction on the underlying blockchain (since the zone file must be modified to include the new hash), making data updates much slower than mutable storage.

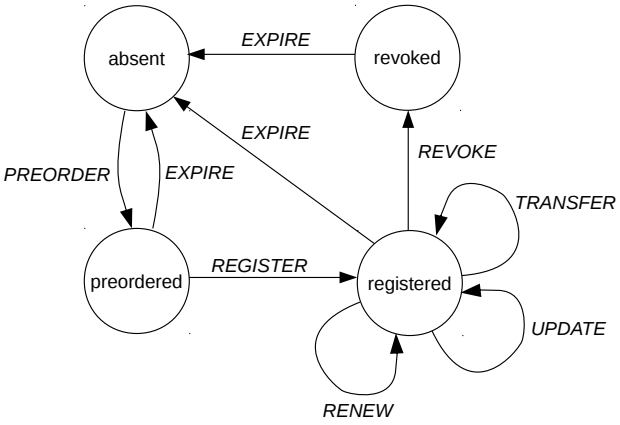


Figure 5: *States and transitions for a name.*

4.4 Naming System

Blockstack uses its four tiers to implement a complete naming system. Names are owned by cryptographic addresses of the underlying blockchain and their associated private keys (e.g. ECDSA-based private keys used in Bitcoin [14]). As with Namecoin, a user *preorders* and then *registers* a name in two steps in order to claim a name without revealing it to the world first and allowing an attacker to race the user in claiming the name. The first user to successfully write both a preorder and a register transaction is granted ownership of the name. Further, any previous preorders become invalid when a name is registered. Once a name is registered, a user can *update* the name-value pair by sending an update transaction and uploading the new value to the storage layer, changing the name-value binding. Name *transfer* operations simply change the address that is allowed to sign subsequent transactions, while *revoke* operations disable any further operations for names.

The naming system is implemented by defining a state machine and rules for state transitions in the *virtualchain*. Figure 5 shows the different states a *name* can be in and how state transitions work. Names are organized into *namespaces*, which are the functional equivalent of top-level domains in DNS—they define the costs and renewal rates of names. Like names, namespaces must be preordered and then registered. Expired names can be re-registered and names can be *revoked* such that they cannot be re-registered for a certain period of time.

4.4.1 Pricing Functions for Namespaces

Anyone can create a namespace or register names in a namespace, as there is no central party to stop someone from doing so. *Pricing functions* define how expensive it

is to create a namespace or to register names in a namespace. Defining intelligent pricing functions is a way to prevent "land grabs" and stop people from registering a lot of namespaces or names that they don't intend to actually use. Blockstack enables people to create namespaces with sophisticated pricing functions. For example, we use the *.id* namespace for our PKI system and created the *.id* namespace with a pricing function where (a) the price of a name drops with an increase in name length and (b) introducing non-alphabetic characters in names also drops the price. With this pricing function, the price of *john.id* > *johnadam.id* > *john0001.id*. The function is generally inspired by the observation that short names with alphabetics only are considered more desirable on namespaces like the one for Twitter usernames. It's possible to create namespaces where name registrations are free as well. Further, we expect that in the future there will be a reseller market for names, just as there is for DNS. A detailed discussion of pricing functions is out of the scope of this paper, and the reader is encouraged to see [31] for more details on pricing functions.

Like names, namespaces also have a *pricing function* [13]. **To start the first namespace on Blockstack, the *.id* namespace, we paid $10,000 in bitcoins to the network. This shows that even the developers of this decentralized system have to follow Blockstack rules and pay appropriate fees.**

4.5 Simple Name Verification

Blockstack nodes can independently calculate a *consensus hash* at any blockchain block. Consensus hashes help Blockstack nodes figure out if they have the same view of the global state at any given block. Each consensus hash $CH(h)$ is constructed from block h's sequence of virtualchain operations V_h, as well a geometric series of prior consensus hashes P_h defined by:

$$CH(h) = hash(V_h + P_h)$$

where

$$P_h = \{CH(h-2^i) | i \in \mathbb{N}, h-2^i >= h_0\}$$

and h_0 is the first block. Other than detecting that two Blockstack nodes have the same global view, consensus hashes also address the *endless ledger problem* (defined in Section 4.1). As the underlying blockchain grows in size, new Blockstack nodes need to process more and more blocks before they boot up.

A new Blockstack node can bootstrap by using an untrusted database of state information at a given block number, combined with a trusted consensus hash $CH(h)$ of the same block number. The block number is also termed the *block height* in the literature, and it increases

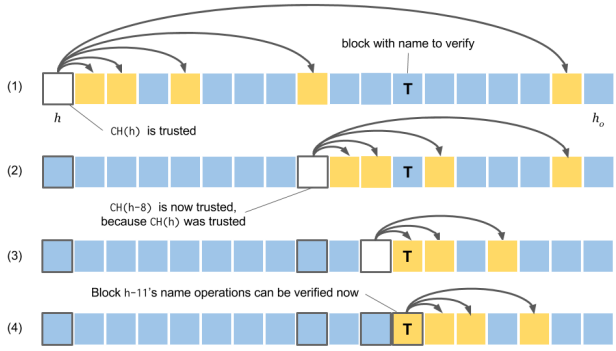


Figure 6: *Overview of SNV. Example SNV query of a record in block* ***T****.*

with each new block. A new Blockstack node can reconstruct the virtualchain from the untrusted database and reprocess virtualchain operations at each blockchain block, recalculating each $CH(h)$ along the way. If the final consensus hash matches the trusted consensus hash at h_n, then the database associated with h_n is trustworthy and the node can start processing blocks after h_n. This is much faster than the traditional approach of starting from the first block h_0 and fetching all transactions, even though most of them will be discarded.

The process of verifying the authenticity of a prior name operation with a later trusted consensus hash is called *Simplified Name Verification (SNV)*. SNV enables support for "thin clients," which can query the past state of the system without running Blockstack nodes or having access to the full blockchain history. Support for thin clients is important for users on mobile devices.

As such, if a user trusts that $CH(h)$ is authentic, then she can query and verify the *virtualchain* operations V_h and previous consensus hash P_h for block h. The construction of $CH(h)$ allows a user to verify the authenticity of any *virtualchain* operation from a block with height $h_{prior} < h$, using only a logarithmic number of queries. Figure 6 shows an example SNV query. Each row represents the blockchain, in decreasing block height order from left to right ($h > h_0$). Here, the user is able to verify the authenticity of a name operation in a target block (marked with a T). In each step, the user recursively trusts the consensus hash for the white outlined blocks.

On current commodity hardware, booting new Blockstack nodes can take 1-2 hours with SNV, compared to 2-4 days without SNV. Further engineering improvements in our Python implementation are currently possible.

4.6 Performance of Reads and Writes

We evaluated the performance of reads and writes through Blockstack to demonstrate that it reads and

Figure 7: *Performance overhead of Blockstack.*

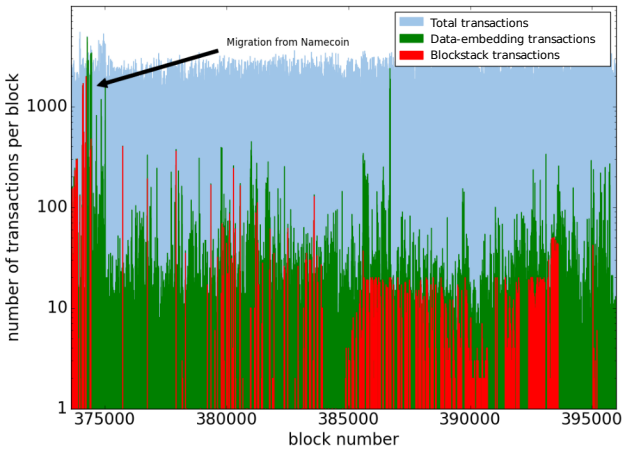


Figure 8: *All data-embedding transactions on Bitcoin; it's already becoming a frequent use case.*

writes files at competitive rates with the underlying storage. Blockstack adds a negligible constant storage space overhead per file (roughly 5% larger files with compression). There is CPU overhead for encryption and compression, but since the file size difference is very small, the network performance for reads and writes is similar to directly accessing the underlying storage service.

The write performance and overheads associated with uploading 1, 10, and 100 megabyte files to Amazon S3 is shown in Figure 7 (each trial was performed 25 times). We see that the CPU-bound overhead is in the order of 2 seconds for large (100MB) files. Many low-hanging performance optimizations still remain in our implementation. Similarly, reading encrypted files from Blockstack with S3 as storage backend is competitive with a direct read from S3 (Figure 7). We omitted the file download time to emphasize the overhead in the graph. The sources of overhead, verifying the signature and decrypting the data, are CPU-bound while in practice performance will largely be network-bound for wide-area usage.

5 Lessons from Migration to Bitcoin

We implemented Blockstack in 40,344 lines of Python code [13] and the current implementation uses Bitcoin as the underlying blockchain. In September 2015, we completed migration of 33,000 users of our production PKI system [43], from Namecoin to Blockstack/Bitcoin. These users were migrated from the *u/* namepsace on Namecoin to the *.id* namespace on Blockstack.

Blockstack embeds additional data in Bitcoin transactions using special fields dedicated for including arbitrary data [12]. Embedding additional data in Bitcoin transactions is already a popular way of defining higher-level protocols on top of Bitcoin, like Counterparty [20], Open Assets [44], etc. Figure 8 shows recent bandwidth usage of data-embedding protocols on the Bitcoin blockchain. The spike of 10,000+ transactions, near block 375000, was during our migration to Bitcoin. Our production system [43] currently accounts for most of Blockstack transactions, which is currently 26.9% of all data-embedding transactions ever made on Bitcoin [45]. Below are some observations we made while working with the Bitcoin network:

Network Throughput: Bitcoin currently supports between 3 and 7 transactions per second with a 1MB block size. Even after a year of heated debate [52] amongst the Bitcoin developers and the broader community, the block size has not been increased. We noticed these limitations first hand when we throttled our transactions so that our transactions wouldn't exceed 20-30% of Bitcoin blocks, which in turn significantly increased the amount of time it took for completing registrations. When scaling to millions of users, as opposed to thousands, even 8MB blocks will not suffice and the community needs to look into performing registrations across multiple chains [10] and novel methods for packing multiple name operations in a single transaction (an area of future work for us).

Network Attacks: During our migration to Bitcoin, a UK-based company called CoinWallet was performing a stress test on the Bitcoin network [17]. The stress test included a high volume of small transactions which had transaction amounts that were too low for miners to package in a block (due to protocol rules designed to prevent spam). This resulted in an extremely high number of unconfirmed transactions on the network and we ended up paying 2-3 times higher transaction fees to get our transactions packaged by miners. This experience shows how

a single actor can force high mining fees on the rest of the network (although in this case there was a cost factor attached to the attack). We believe that networking attacks on blockchains, like the one we experienced or other DDoS attacks [37], are likely to become more frequent. Protections against such attacks is an important area of future research.

6 Related Work

Binding names to values in naming systems is a well-explored problem space. UIA [24] gives a great overview of global naming systems and their importance. We encourage the reader to UIA [24] for a detailed background on naming systems. Unlike Namecoin [41] or Blockstack, UIA doesn't try to provide globally unique names. In authentication systems like InCommon [27], OpenID [47], and the Web's certificate authorities, a federation of authorities attests to bindings. Blockstack, however, does not require a federation.

Other than Namecoin, blockchains like Ethereum [7] and BitShares [5] also have support for human-readable names. Further, sidechains [10] enable implementation of naming systems as an alternate blockchain that is linked to the main Bitcoin blockchain. All these designs involve smaller, alternate blockchains and Blockstack directly uses the most secure blockchain (Bitcoin). Non-blockchain based PKI systems, like Keybase [34] and CONIKS [38], achieve some of the same goals as Blockstack for automating key management. The main difference is that Blockstack both provides users with direct access and control over their data, without placing trust in any specific principals while giving global state.

In networked systems it's hard to get global state without involving central trusted parties [33], Blockstack is able to provide global state (and not just approximate global state). Our system is open ("permissionless"), whereas existing wide-area systems like OceanStore [21] and Bonafide [16] have a closed ("permissioned") set of peers that use BFT agreement to make progress for the whole system. Blockstack differs from decentralized storage systems which allow open membership but offer stronger-than-eventual data consistency (like Shark [9], Pond [48], and Scatter [25]) by focusing on decentralization while supporting a wide variety of external datastores that give strong consistency.

Storage-oriented cryptocurrency blockchains like Filecoin [23], Permacoin [40], and Storj [50] seek to replace cloud storage by distributing files as sets of transactions within a blockchain, and rewarding miners for proof-of-storage (instead of proof-of-work). Blockstack differs from these systems by decoupling hosting data from operations of the underlying blockchain, allowing developers to use storage systems appropriate for their problem domains. Blockstack currently uses a simple Kademlia [36] based DHT as discovery layer, but other protocols like Chord [51] or caching optimizations like Beehive [46] are possible.

7 Conclusion

Our experience with running a production network on Namecoin, one of the oldest and largest cryptocurrency blockchains other than Bitcoin, shows how a single miner consistently had more than 51% hashing power and how network reliability was far inferior to Bitcoin. Our data shows that out of the hundreds of blockchains currently in use [6], even the more stable and more popular blockchains like Namecoin are not suitable for production use. Currently, the security of Bitcoin far outweighs other blockchains.

We have presented Blockstack, a blockchain-based naming and storage system. Blockstack introduces separate control and data planes, and by doing so, it enables the introduction of new functionality without modifying the underlying blockchain. The design of Blockstack was informed by a year of production experience from one of the largest blockchain-based production systems to date. We have made several novel improvements (like introducing the ability to do cross-chain migrations, faster bootstrapping of new nodes, and keeping data updates off the slow blockchain network) that make it easier to build decentralized services using publicly-available infrastructure. Our performance results show that Blockstack can give comparable performance to the underlying storage service and only introduces a small CPU overhead. We've released Blockstack as open-source [13].

Acknowledgments

We thank Bitcoin developers Gavin Andresen for first pushing us to build a Namecoin-like system on top of Bitcoin, Matt Corallo for feedback on protocol design, and Peter Todd for discussion around delaying rewards to miners and associated security implications. Namecoin developer Daniel Kraft helped us with debugging Namecoin issues, Ryan Castellucci with getting mining stats, Guy Lepage with generating performance graphs, and Riccardo Casatta with collecting stats on data-embedding transactions. We also thank Larry Peterson, Lakshmi Subramanian, Andrea LaPaugh, our shepherd Mohit Aron, Jon Howell, and anonymous USENIX ATC reviewers for helpful discussions and feedback. Finally, we'd like to thank the entire Blockstack open-source community (`http://blockstack.org`) for various discussions, bug reports, and Github pull requests during the production-ready release of Blockstack.

References

[1] Bitcoin hashrate. https://blockchain.info/charts/hash-rate.

[2] Bitcoin transactions per blocks. https://blockchain.info/charts/n-transactions-per-block.

[3] Blockstack ID format, version 2. https://blockstack.org/docs/blockstack-profiles.

[4] Crypto-currencies stats – active nodes. https://bitinfocharts.com.

[5] Bitshares namespaces, 2016. http://docs.bitshares.eu/namespaces/index.html.

[6] Cryptocurrency market cap, Jan. 2016. http://www.coincap.io.

[7] A next-generation smart contract and decentralized application platform, 2016. https://github.com/ethereum/wiki/wiki/White-Paper.

[8] Adam Back. Hashcash - A Denial of Service Counter-Measure. Tech report, 2002. http://www.hashcash.org/papers/hashcash.pdf.

[9] S. Annapureddy, M. J. Freedman, and D. Mazieres. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd NSDI*, Boston, MA, 2005.

[10] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timon, and P. Wuille. Enabling Blockchain Innovations with Pegged Sidechains. White paper, Blockstream, 2014. https://blockstream.com/sidechains.pdf.

[11] Bitcoin Core Developers. Bitcoin core version 0.10.0 release notes: Consensus library, Feb. 2015. https://bitcoin.org/en/release/v0.10.0.

[12] Bitcoin.org: Bitcoin developer guide, 2015. http://bitcoin.org/en/developer-guide.

[13] Blockstack source code release v0.10, 2016. http://github.com/blockstack/blockstack-server.

[14] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 104–121, 2015.

[15] Chainpoint white paper. https://tierion.com/chainpoint.

[16] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Tiered fault tolerance for long-term integrity. In *FAST*, pages 267–282, 2009.

[17] Coindesk. Bitcoin network stress test could occur next week, Sep 2015. http://coinde.sk/1Ku5oWc.

[18] Coindesk. State of bitcoin 2015: Ecosystem grows despite price decline, 2015. http://coinde.sk/1tJDDvv.

[19] Coindesk. State of blockchain q1 2016: Blockchain funding overtakes bitcoin, May 2016. http://www.coindesk.com/state-of-blockchain-q1-2016/.

[20] Counterparty protocol specifications. http://counterparty.io/docs/protocol_specification/.

[21] P. Eaton, H. Weatherspoon, and J. Kubiatowicz. Efficiently binding data to owners in distributed content-addressable storage systems. In *Security in Storage Workshop, 2005. SISW'05. Third IEEE International*, pages 12–pp. IEEE, 2005.

[22] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. *CoRR*, abs/1311.0243, 2013.

[23] Filecoin: A Cryptocurrency Operated File Network. Tech report, 2014. http://filecoin.io/filecoin.pdf.

[24] B. Ford, J. Strauss, C. Lesniewski-Laas, S. Rhea, F. Kaashoek, and R. Morris. Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, Nov. 2006.

[25] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[26] M. Gronager. Namecoin was stillborn, i had to switch off life-support, Oct 2013. https://bitcointalk.org/index.php?topic=310954.

[27] Incommon federation. https://www.incommon.org/federation/.

[28] D. Johnson, A. Menezes, and S. A. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Sec.*, 1(1):36–63, 2001.

[29] Juan Benet. IPFS - Content Addressed, Versioned, P2P File System. Draft, ipfs.io, 2015. https://github.com/ipfs/papers.

[30] Jude Nelson and Larry Peterson. Syndicate: Virtual cloud storage through provider composition. In *ACM BigSystem*, 2014.

[31] H. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan. An empirical study of Namecoin and lessons for decentralized namespace design. *WEIS '15: Proceedings of the 14th Workshop on the Economics of Information Security*, June 2015.

[32] D. Kaminsky. Spelunking the triangle: Exploring aaron swartzs take on zookos triangle, Jan 2011. http://dankaminsky.com/2011/01/13/spelunk-tri/.

[33] S. Keshav. Efficient and decentralized computation of approximate global state. *SIGCOMM Comput. Commun. Rev.*, 36(1):69–74, Jan. 2006.

[34] Keybase. `https://keybase.io`.

[35] J. A. Kroll, I. C. Davey, and E. W. Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *WEIS 2013*.

[36] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[37] J. McGovern. Official statement on the last weeks ddos-attack against ghash.io mining pool, 2015. `http://bit.ly/1nu49vR`.

[38] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman. CONIKS: bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[39] Merge mining specification. `https://en.bitcoin.it/wiki/Merged_mining_specification`.

[40] A. Miller, A. Juels, E. Shi, B. Parno, and J. Katz. Permacoin: Repurposing bitcoin work for data preservation. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 475–490. IEEE, 2014.

[41] Namecoin. `https://namecoin.info`.

[42] Namecoin core, required software for miners, 2015. `https://github.com/namecoin/namecoin-core`.

[43] Onename. `https://onename.com`.

[44] Open assets protocol. `http://www.openassets.org`.

[45] Statistics of usage for bitcoin op_return. `http://opreturn.org`.

[46] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

[47] D. Recordon and D. Reed. Openid 2.0: A platform for user-centric identity management. In *Proceedings of the Second ACM Workshop on Digital Identity Management*, DIM '06, pages 11–16, New York, NY, USA, 2006. ACM.

[48] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, volume 3, pages 1–14, 2003.

[49] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Tech report, 2009. `https://bitcoin.org/bitcoin.pdf`.

[50] Shawn Wilkinson and Tome Boshevski and Josh Brandof and Vitalik Buterin. Storj: A peer-to-peer cloud storage network. Tech report, storj.io, 2014. `http://storj.io/storj.pdf`.

[51] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *IEEE Transactions on Networking*, 11, Feb. 2003.

[52] A. V. Wirdum. A closer look at bip100: The block size proposal bitcoin miners are rallying behind, Aug. 2015. `http://bit.ly/1VbyoXP`.

[53] P. R. Zimmermann. *The Official PGP User's Guide*. MIT Press, Cambridge, MA, USA, 1995.

Satellite: Joint Analysis of CDNs and Network-Level Interference

Will Scott, Thomas Anderson, Tadayoshi Kohno, and Arvind Krishnamurthy
{wrs, tom, yoshi, arvind}@cs.washington.edu

Abstract

Satellite is a methodology, tool chain, and data-set for understanding global trends in website deployment and accessibility using only a single or small number of standard measurement nodes. Satellite collects information on DNS resolution and resource availability around the Internet by probing the IPv4 address space. These measurements are valuable in their breadth and sustainability - they do not require the use of a distributed measurement infrastructure, and therefore can be run at low cost and by multiple organizations. We demonstrate a clustering procedure which accurately captures the IP footprints of CDN deployments, and then show how this technique allows for more accurate determination of correct and incorrect IP resolutions. Satellite has multiple applications. It reveals the prevalence of CDNs by showing that 20% of the top 10,000 Alexa domains are hosted on shared infrastructure, and that CloudFlare alone accounts for nearly 10% of these sites. The same data-set detects 4,819 instances of ISP level DNS hijacking in 117 countries.

1 Introduction

After several generations of elaborate measurement platforms, it remains difficult to characterize how web content is distributed and the extent to which it is open and unfettered. This lack of understanding is reflected in the questions we cannot easily answer: Which countries have servers operated by Google or Microsoft, and what is the footprint of various content distribution networks (CDNs)? Which websites have degraded availability due to network interference? Which sites are powered by various CDNs such as CloudFlare or Fastly? Which ISPs run caching proxies or other stateful middle-boxes? And so on.

Measurements that characterize web access is of value for publishers and end users alike. For publishers, it allows for informed choice about how to locate their content – which CDNs to use, and appropriate trade-offs between security and connectivity. Website optimization is dependent on the expected latency of user connections, which is difficult for publishers to predict in advance of choosing hosting or providers. For users, these measurements provide insight into whether operators comply with local regulations, and which sites will be able to warehouse data within their jurisdiction. Transparency in network censorship (what sites are being blocked, and how) is also critical to regulatory oversight and informing public debate. These lists are almost always kept secret, and even when available, it is difficult for watchdogs to verify that they reflect reality. This opacity makes it difficult for users to advocate for changes in policy or trust existing systems.

While we have some understanding of what measurements can address these questions, there is no existing data set or measurement platform that holds the answers. In fact, there are many challenges both in collecting the measurement data and analyzing it to characterize the current state of web content distribution.

First, we need measurements from globally distributed vantage points in order to characterize global website accessibility. Understanding both the Internet-wide expansions of CDNs and the interference practices around the world requires data from a diverse set of ISPs. Such a measurement platform does not exist yet in a public and transparent manner that supports reproducibility of results. Distributed collection platforms also face concerns of retribution towards those hosting the measurements.

Second, since deployment and accessibility characteristics can change rapidly, collection must be fine-grained and timely. Many documents of interference choose a single fixed point in time, or provide yearly updates. To be effective, we need to provide alerts that policies have changed quickly, which raises questions regarding how many domains can be monitored and what granularity is sufficient.

Third, the analysis of how websites employ CDNs and the identification of network interference must be tackled jointly. For example, when ISPs block websites by redirecting them to a block page, those servers are easily misconstrued as a CDN node for that geographical region. Consider the example of twitter.com. As shown in Table 1, the domain resolves to different IPs in the US, Russia, and China. A naive CDN mapping would conclude that there are likely points of presence in all three countries, while a naive interference measurement might conclude interference in both China or Russia, or might give up due to the diversity of IPs returned. In reality, the Russian IP maps to a Twitter CDN node, while

Resolver	Response	Behavior
USA (8.8.8.8)	199.59.149.198	Twitter
Russia (77.88.8.8)	199.16.156.102	Twitter
China (180.76.76.76)	159.106.121.75	Failure

Table 1: Resolutions of Twitter.com by different resolvers

the Chinese resolution is due to interference.

In the remainder of this paper, we demonstrate Satellite, an automatic framework that is able to identify CDN infrastructure in tandem with anomalies. Through the use of Internet scanning and reflecting queries on public infrastructure we avoid the pitfalls that come with distributed infrastructure. By clustering sites based on DNS resolution, and by finding responses which do not fit those clusters, we are able to identify interference without misclassifying CDNs. This analysis is able to both monitor the growth of shared hosting platforms, and which sites are blocked by network-level interference.

We address the need for a distributed measurement platform by using a single end-host to collect DNS resolutions from a large number of globally-distributed and open DNS resolvers. Instead of pursuing crowd-sourced deployments or analyzing limited snapshots of data obtained from operators in privileged positions, we instead focus on what is possible from active measurements conducted by a single end-host. Doing so both reduces the barrier to entry for organizations to run their own independent measurements, and removes the complex work of coordinating a distributed testbed and verifying the untrusted dataset collected from it. While results from a single machine may be biased, the validation steps are the same as those of distributed infrastructure; it continues to be the case that you can't definitively claim interference from a single instance of connection failure, and instead extract evidence from aggregate trends.

We choose DNS as our main platform of measurement because it has developed as a narrow waist that is used both by CDNs for routing traffic and for the interposition of block pages by ISPs and nations. CDNs use the DNS resolution process for load balancing and routing because it is the first step in a web page access; making a good decision at the DNS level ensures fast connections for the rest of the loading process. Network interference also often occurs at the DNS layer, because while a single IP may host content for many sites, DNS requests have an easily parseable format and allow restriction of specific domains. The existence of shared infrastructure that hosts many sites (CDNs) is exactly why interference continues to be commonly implemented at a resolution level.

We address the need for timely global measurements by designing a system that can measure the global connectivity of tens of thousands of domains with weekly precision. By focusing on coverage rather than a specific event or geographic region, Satellite acts as a database supporting higher level analysis of policy changes as they occur. By measuring the Alexa top 10,000 global domains, we are able to detect evidence of interference in many countries and automatically detect most popular shared infrastructure without manual targeting of measurement.

We address the need for joint understanding of infrastructure and interference through our algorithmic interpretation of Satellite data. We correlate the addresses of domains across ISPs and learn the customer pools of CDNs. Looking at the pools of IPs, we can learn the points of presence of CDNs and which CDNs have business relationships with which ISPs. By looking at which locations resolve to which points of presence we can understand the geographic areas served by different points of presence. By tracing the patterns of divergence from clusters, we are able to separate the effects of network interference from confounding site distribution factors.

Satellite has limitations in the view of Internet infrastructure it reveals. Some shared services explicitly partition incoming requests across disjoint sets of servers. Dedicated IP addresses are used to support SSL for some old browsers, to reduce dynamic generation of certificates, and as part of fault isolation strategies. Akamai is an example of such a shared infrastructure. Satellite does not report these platforms as single entities, but rather as multiple smaller shards, defining the more specific subsets of IPs assigned to each customer (i.e., domain).

Satellite is a fully open project consisting of the code for data collection and analysis, a growing year-long repository of collected data, and derived views of site structure and interference. Satellite is built for transparency, minimizing the trust that needs to be placed in the system or its operators. We are working with several independent organizations to independently collect and corroborate data. We hope this structure enables future researchers to trust collected data without the need to replicate collection work. Building Satellite to run on a single machine is aimed to maximize the sustainability of the project, and our ability to amass a longitudinal data set of changing Internet behavior.

The major contributions of Satellite are:

- A single-node measurement system for monitoring global trends in network interference and CDN deployment.
- An algorithm for the joint analysis of network anomalies and determination of shared infrastructure from point measurements of domain resolution.
- Data on the distribution and accessibility of 10,000 popular domains over the last two years.

In the remainder of this paper we will elaborate on the design and operation of Satellite, and present some of the site behavior we have discovered through these measurements.

2 System Design

Satellite is motivated by a number of explicit design goals differentiating it from existing platforms and systems.

- **External Data Collection**: We want the system to function without requiring in-situ resources. This avoids the need to recruit volunteers, and focuses on safety and coverage across networks.
- **Continuous Measurement**: We want the system to be able to quickly notice changes in CDN deployments and network access policies.
- **Transparent and Ethical Measurements**: We want the system to be transparent, so that others can easily trust and make use of collected data. We aim for high ethical standards to minimize harm to DNS server operators from collected data.
- **Joint analysis of CDN deployments and Network Interference**: We want a system which simultaneously measures shared infrastructure and interference of web access, since the two are tightly intertwined.

2.1 System Overview

The Satellite system is arranged as a pipeline which collects and analyzes data. It is run as a weekly cron job, which schedules data collection, and performs initial aggregation, analysis and archiving of each data set. The implementation details of the pipeline are described in more detail in Section 3. At a high level, Satellite is structured into the following discrete tasks:

Identifying DNS resolvers by scanning the Internet. We detect active, open, long-lived DNS resolvers through active probing.

Assembling a target domain list by expanding a list of popular domains to ensure CDN coverage.

Performing active DNS measurements where candidate domains are measured against discovered resolvers.

Collection of supplemental data where organization metadata and geolocation hints are gathered.

Aggregation of DNS resolutions by combining records at the AS level to allow for efficient processing in subsequent analysis.

Joint analysis of CDNs and network interference through the calculation of fixed-points in clusters of domains believed to use shared infrastructure.

Export of measurement results by publishing visualizations and data sets with footprints of CDNs and significant observed anomalies.

2.2 Identifying DNS resolvers

Our measurements are based on gathering data on how domains behave for different clients around the world. There are several options available for this type of collection. Traditionally, researchers have used cooperating hosts in a variety of networks [24, 30]. More recently, the EDNS extension has allowed clients to indicate that they are asking for a response that will be used by someone in a different geographic area to approximate multiple vantage points [35, 6]. Very few domain name servers support EDNS, but we can take advantage of the same behavior the mechanism is designed to fix. By making requests to many resolvers, we can learn the different points of presence for target domains. For instance, the `8.8.8.8` resolver is operated by Google and provides a US-centric view of the world, while `180.76.76.76`, "BaiduDNS", provides a Chinese centric view.

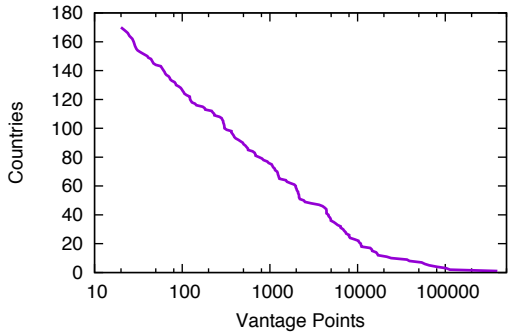


Figure 1: DNS servers discovered in each Country. We find 169 countries hosting DNS resolvers in more than 20 class-c networks.

We enumerate the IPs acting as DNS resolvers by probing the IPv4 Address space with zmap [12]. Compared to 32 million open DNS servers monitored by the Open Resolver Project [28] (a service measuring the potential for reflected denial of service attacks through DNS, it does not share the IPs of discovered servers), we independently discover 12 million servers which respond to requests with a well-formed response. Of these, 7 million servers across 1.5 million class-c (/24) networks offer recursive resolution and give a correct IP address when asked to resolve our measurement server. These servers provide coverage of 20,000 ASes (Autonomous Systems, typically representing an ISP), and cover 169 countries with at least 20 class-c networks, as shown in Figure 1.

2.3 Ethics of Collection

Our measurements prompt machines in remote networks to resolve domains on our behalf. This traffic to remote networks may result in unintended consequences to these relays, and as such we do our best to minimize harm in keeping with best practices [11].

Open DNS resolvers are a well known phenomenon, and lists of active resolvers can be downloaded without the overhead we incur in scanning. We find that the act of scanning the IPv4 address space to find active resolvers does generate abuse complaints from network operators. By maintaining a blacklist of networks which have requested de-listing (less than 0.5% of the address space), we have not received any complaints related to our scan-

ning or subsequent resolutions in the last quarter. Some operators have asked us to keep their network spaces private, which prevents us from releasing this list publicly. Others running the system should expect to recreate a similar list. We have never received a complaint from overloading a DNS resolver with queries for our tracked domains.

We abide by the 7 harm mitigation principles for conducing Internet-wide scanning outlined by the zmap project [12]. In particular, we (a) coordinated with the network administrators at our university in handling complaints, (b) ensured we do not overload the outbound network, (c) host a web page explaining the measurements with an opt-out procedure, and have clear reverse DNS entries assigned to the measurement machine, (d) clearly communicate the purpose of measurements in all communications, (e) honor any opt-out requests we receive, (f) make queries no more than once per minute, and spread network activity out to accomplish needed data collection over a full one-week period, and (g) spread the traffic over both time and source addresses allocated to our measurement machine.

To get a better sense of the impact our queries have on resolvers, we operated an open DNS resolver. In a 1 week period after running for 1 month, the resolver answered over one million queries, including 800,000 queries for domains in the Alexa top 10,000 list. Satellite made only 1,000 of these requests.

We have additionally adopted a policy of only probing DNS servers seen running for more than one month to reduce the potential of sending queries to transient resolvers. This reduces our resolver list by 16%[1]. Measurements in IP churn indicate that the bulk of dynamic IPs turn over to subsequent users on the order of hours to days, making it unlikely that our measurements target residential users [38].

2.4 Mapping CDNs and Network Interference

We know that for many CDNs it is common to resolve domains to different IP addresses based on where the client is. While the diversity of IPs makes it more difficult to understand what an 'unexpected' deviation is, the primary insight we can use is that in many cases these CDN infrastructures are shared by many sites. The set of sites on a shared infrastructure is often independent of the set of sites which are targeted by network interference.

Consider the case of `thepiratebay.se`, a domain hosted with `strawpoll.me` on Cloudflare. In a US location, like the DNS resolver operated within UC Berkeley (AS25), both domains resolve to IPs in the `141.101.118/24` subnet. However, across many networks in Iran (for instance AS50810), the first resolves instead to `10.10.34.36`, an internal LAN address, while the second continues to resolve to Cloudflare owned IPs.

[1] Specifically comparing the live resolvers discovered between March 20th and April 20th, 2015.

$domains \leftarrow$ the set of all domains
$ips \leftarrow$ the set of resolved IPs
function EDGE($domain, ip$)
  return |ASes where $domain$ resolved to ip|
end function
function IPTRUST($domain, ip$)
  $\triangleright\ 0-1$ value representing how likely an IP is a server for a domain.
  return $\frac{\sum_{d \in domains} \text{EDGE}(d,ip) * \text{DOMAINSIMILARITY}(domain,d)}{\sum_{d \in domains} \text{EDGE}(d,ip)}$
end function
function WEIGHT($domain, ip$)
  $\triangleright$ EDGE weighted by IPTRUST.
  return $\text{EDGE}(domain, ip) * \text{IPTRUST}(domain, ip)$
end function
function DOMAINSIMILARITY(dom_a, dom_b)
  $\triangleright\ 0-1$ value representing how likely two domains are hosted on the same servers.
  return

$$\frac{\sum_{ip \in ips} \text{WEIGHT}(dom_a, ip) * \text{WEIGHT}(dom_b, ip)}{\sqrt{\sum_{ip \in ips} \text{EDGE}(dom_a, ip)^2} * \sqrt{\sum_{ip \in ips} \text{EDGE}(dom_b, ip)^2}}$$

end function

Figure 2: Pseudocode of CDN and interference detection joint analysis algorithm. The two functions `DomainSimilarity` and `IPTrust` depend on each other, and are iteratively computed until a fixed point is approximated. The result of these two functions allows direct determination of both the IPs hosting clusters of domains, and the resolutions which are anomalous.

To automate this form of detection, we automatically find cliques of domains hosted on the same infrastructure, and use the combined resolutions of those domains to map the IPs of the underlying infrastructure. Using multiple domains will help us to overcome the randomness present in individual domain resolutions, and to notice when one domain behaves strangely in a specific geographic region. We are not using IP metadata to map provider infrastructure, but rather the sets of IPs (potentially across providers) that form the footprints of popular domains.

To process the data, we perform a joint analysis using the algorithm in Figure 2 (also described in text below). Then, we use the stable values from that computation to extract cliques and deviations, which represent shared infrastructure and interference respectively.

2.4.1 Joint Analysis Algorithm

Given a bipartite graph linking IP addresses and domains, our goal is to separate the graph into two components: 'real infrastructure', and 'interference'. An intuition of how to think of this separation is shown in Figure 3. To find this separation, we compute two quantities: A similarity metric `DomainSimilarity`, for how close

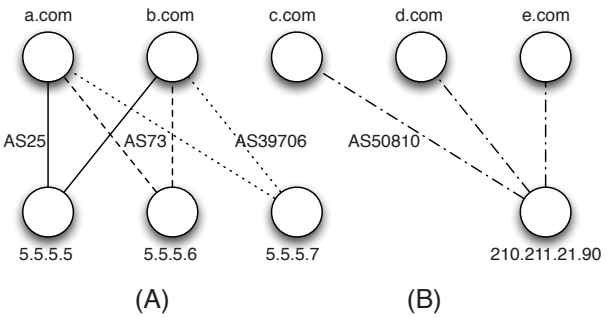


Figure 3: The relationship between Domains and IPs. Each edge corresponds to a resolution, labeled by the autonomous system of the resolver. In this example, we see *a.com* resolve to `5.5.5.5` in UC Berkeley, AS 25. In (A) we see a clique of domains supported by the same infrastructure, while (B) shows otherwise unrelated domains resolving to the same IP within AS 50810.

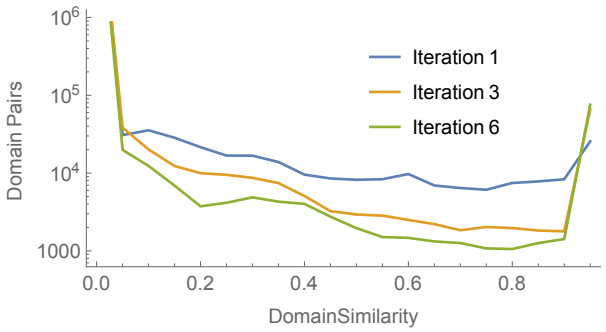


Figure 4: `DomainSimilarity` distribution after iterative calculation. After the first iteration, 25,000 edges with similarity above 95% are found. After 5 iterations there are 75,000 strong similarities, fewer 'uncertain' similarities, and less than 1000 similarities that changed by more than 5%.

two domains are, and a trust metric `IPTrust`, for how likely an IP is to be an authentic resolution for a given domain. In Figure 3a, we would hope that *a.com* and *b.com* have a high `DomainSimilarity`, since they resolve to the same IPs. In Figure 3b, we would hope the IP `210.211.21.90` has a low `IPTrust` score, since many otherwise unrelated domains resolve to it. This process is similar to the HITS algorithm for finding "authoritative" sources for pages [23].

The `DomainSimilarity` metric specifically represents the fraction of the time that two domains resolve to the same IPs. We use the different IPs as independent dimensions in which the resolutions of each domain can be represented as a vector. The distance between Domains is then the cosine distance between the two resolution vectors.

The `IPTrust` metric calculates the confidence for whether any given IP address resolution of a domain is correct. To calculate our confidence in a resolution, we say the probability a domain resolves to an IP is equal to the average similarity between that domain and the other domains which have resolved to that IP. To score whether we believe that `thepiratebay.se` resolves to `10.10.34.36`, we would look at other domains which have resolved to `10.10.34.36` and consider their `DomainSimilarity` with `thepiratebay.se`.

We now discuss cases where a provider allocates non-disjoint but partially overlapping sets of IPs to different domains. For example, if a domain `a.com` resolves to IPs A,B, and C, while `b.com` resolves to C, D, and E. If the different IPs are in the same class-c network, then our analysis will see both `a.com` and `b.com` as resolving to the class-c network that corresponds to A, B, C, D, and E, thus attributing a high `IPTrust` value to the class-c network for the two domains. Class-c is chosen as the most specific public announcement of IP ownership, limiting accidental grouping of different providers. If the IPs are in different class-c networks, the `IPTrust` can still be high if the `DomainSimilarity` is high. In cases where there is only a small fraction of IP space overlap, metadata is not present, and `DomainSimilarity` is low, Satellite will consider the two domains to be in separate clusters. This will attribute a low `IPTrust` to C.

For intuition behind these metrics, consider the representative case of the Fastly CDN. Taking one IP range, `23.235.47.0/24`, we find that across the 72 domains Satellite clusters as Fastly the `IPTrust` metric ranges between 0.75 and 0.98. Across all other domains, this IP range was seen as a resolution for 22 other domains, across which its average `IPTrust` was 0.20 and maximum was 0.30. The range of `IPTrust` in the Fastly cluster shows that the strongly connected cluster boosted scores for IPs that were infrequently resolved for some domains.

To derive an initial estimate of `DomainSimilarity`, we set `IPTrust` to 1.0. We then iteratively calculate these two quantities until a fixed point is approximated, generally in 5-6 iterations. Figure 4 shows the effect of iteration on the distribution of domain similarities. Without iterating to the fixed point, many domain pairs have a similarity coefficient close to 0.5. Subsequent iterations concentrate the emergent clusters to more clearly define shared infrastructure (close to 1.0) as well as block-pages (close to 0.0).

2.4.2 Cliques and Deviations

`DomainSimilarity` and `IPTrust` form the core metrics we need to determine both CDN footprints (the cliques of similar domains and associated set of IP addresses they're served from), and network anomalies (sets of domains sent to IPs with low trust in isolated ASNs).

CDN cliques: To find clusters of domains with similar resolutions in the matrix of calculated `DomainSimilarity` values, we use a greedy

Domain	Alexa Rank
www.ebay.com	18
cntv.cn	79
indiatimes.com	110
dailymail.co.uk	114
etsy.com	149
cnet.com	151
deviantart.com	168
forbes.com	175

Table 2: The highest ranked domains identified in the largest 'Akamai' cluster.

CDN	Size	Representative Domain
CloudFlare	726	reddit.com
Amazon AWS	647	amazon.com
Akamai	410	ebay.com
Google	141	google.com
Dyn	112	webmd.com
Rackspace	77	wikihow.com
Fastly	72	imgur.com
Edgecast	68	soundcloud.com
Incapsula	55	wix.com
AliCloud	54	163.com

Table 3: Largest CDN clusters. The top 10 CDNs account for 20% of monitored domains.

algorithm of first making arbitrary clusters, and then finding the best 'swaps' possible until a local maxima is found [13]. This clustering technique has been found to perform close to human labeling.

Table 2 shows an example of the highest popularity sites that were clustered into the clique representing the Akamai infrastructure. The largest clusters are shown in Table 3. We count the 10 largest shared hosting platforms hosting 1967 domains, making up almost 20% of those measured.

At a global level, strongly connected components represent domains hosted by the same servers. This may be domains resolving to one IP everywhere, or domains with the same CDN configuration which consistently resolve to the same IPs from different vantage points. If we narrow our consideration to the ASes based in a single country, blocking can also appear as a cluster with the block page IP clustered with all of the blocked domains. These clusters are only found in the ASes of individual countries, and the difference between detected clusters globally and nationally is a strong signal for this behavior. On the other hand, this is only one of many ways to interfere with DNS. Some forms, like the response of random IPs used by some Chinese ISPs [5], will reduce `IPTrust` without creating these obvious clusters.

It should be at first surprising that Akamai, one of the largest CDN providers, is represented by a low number of domains. We find that while Akamai transfers a large amount of traffic, we count many of their domains as independent entities for two reasons. First, Akamai often delivers home pages as a relatively small set of IP addresses that are dedicated to HTTPS for the specific customer. Second, Akamai is located in over 1000 different ISPs, with most IPs assigned to servers advertised and using those IPS's AS numbers as their origins. These two factors cause many Akamai customers to be treated as independent entities by Satellite, and not seen as part of their shared serving infrastructure.

We can compare the relationship Akamai has with customers to that of Cloudflare, which also provides 'white-label' services for large customers to customize their presence through custom DNS name servers and SSL deployed for older clients unable to perform server name identification. Cloudflare partitions its customers across several distinct IP spaces. Some of these IPs have reverse PTR and whois information identifying them as Cloudflare, while others do not. The use of IP addresses within Cloudflare ASes and Cloudflare associated WHOIS information allow satellite to cluster these services as one entity with more certainty than the less obviously related Akamai customers.

Network interference: The question of "who is blocking what?" can be answered by finding ASes where a majority of resolutions have low `IPTrust` for a given domain. There are actually several ways in which an AS can deviate, which correspond to different forms of interference. For example, Iran regularly sends `thepiratebay.se` to `10.10.34.36`, and we see `IPTrust` of 6.6×10^{-9} for those resolutions, since the IP is also seen for a number of other blocked domains which don't otherwise overlap.

To extract instances of interference that are reflected in the `IPTrust` metric, we look at the distribution of values for resolutions at the AS level. When the distribution for an AS is depressed in a statistically significant manner (we currently look for a mean 4 standard deviations below the overall distribution) we consider the AS-domain pair to be 'suspicious'.

There are several types of interference which can all be easily distinguished from normal behavior, but which require special identification. We handle these through a decision tree, which provides a conservative estimate of known forms of interference. Crucially, this approach benefits from the fact that we are able to point to the mechanism which triggers each flagging. The categories we classify as interference are:

1. Too few resolutions or too many unparsable responses are received.

2. A domain which is otherwise 'single-homed' (meaning a single IP address is found regardless of client location) resolves to non-standard locations.

3. A domain with an otherwise 'dominant' AS resolves to many ASes.

4. Resolution deviates from an expected CDN cluster.

Instances of interference are accounted to occur because of the first of these classes which is applicable. All of these classes can be inferred from the already computed resolution data. Our initial AS-level aggregation allows us to directly find invalid or suppressed resolutions. We showed in Figure 5 that the majority of the most popular domains are single-homed, which we use for the third and fourth decisions. Finally, for domains which appear to be hosted on shared infrastructure, we use the `IPTrust` score computed above. When resolutions deviate from the expected CDN footprint, we are able to include automatic analysis of the availability of the most high-profile instances of interference.

3 Implementation

3.1 Assembly of domain list

To understand how sites behave, we must first know the sites we are interested in monitoring. It is unrealistic to monitor all domains on the Internet, since there are technically an infinite number of registered domains due to the dynamic nature of sub-domain resolution. Without a priori knowledge of CDNs and their expected IPs around the world, we need to monitor a representative set of domains to organically learn that knowledge.

We accomplish this goal by targeting the top 10,000 worldwide domains as measured by Alexa[2]. All of these domains receive high amounts of traffic. The least popular, `qualcomm.com`, is estimated to receive over 10,000 visitors per day. While not a perfect list, 10,000 domains contains the diversity needed to organically discover important CDNs. Looking at the smaller Alexa top 1k domain subset, we find that under a quarter of the domains we cluster into CDNs are listed. For services like CloudFlare, which partition their IP space across different domains, our clustering algorithm would be overly cautious without access to an appropriately diverse sample set.

We make HTTP requests to each domain, since we find that many bare domains (e.g. `expedia.com`) redirect to a prefixed domain (e.g. `www.expedia.com`), which are served on different infrastructure. When we detect such redirections, we include both the bare and prefixed domains in subsequent steps. We observe these redirects in roughly one fourth of monitored domains.

3.2 Active DNS measurement

Our goal in Satellite is to provide a tool for longitudinal mapping of the accessibility and distribution of web entities. To quickly detect updates and policy changes, we must constrain the amount of time we are willing to allow probing to run. Given the goal of weekly measurements of 10,000 domains from a single host, we request each domain from 1/10th (or roughly 150,000) of discovered DNS vantage points, maintaining geographic diversity while spreading network load across available hosts. This results in a measurement period of roughly 48 hours at a probe rate of 50,000 packets per second. We find our measurement machine to be CPU limited at about 100,000 packets per second. Unlike a typical zmap scan, our resolution probes have a high response rate, which results in significant CPU processing work.

Our probing is accomplished by extending zmap with a custom 'udp_multi' mode, where hosts are sent one of several packets. The packet sent is chosen based on the destination IP address only, resulting in a stable set of requests across measurement sessions — the same resolvers will receive the same queries each week. This approach was chosen for efficiency, multiple scanning processes and accompanying pcap filters increased CPU load and resulted in dropped packets. Instead, we found this extension to be a conceptually simple and efficient extension to the existing zmap tool.

The result of a 48 hour collection process is a 350GB directory containing tuples of resolver IPs, queried domain, time-stamp, and received UDP response. We record the full packet responses we receive, under the assumption that in the future we may find other fields of the DNS responses to be of interest. The raw format of base-64 encoded packets is extremely verbose, but since the response packets for each domain are largely the same, a full run can be compressed to 20GB. By taking this relatively easy step of compression, our measurement machine has had no trouble storing our year of collection results.

3.3 Supplemental data collection

There are several pieces of supplemental data that are valuable to Satellite in understanding the measurements we conduct. For IP addresses of interest, we collect information to improve our ability to map IPs back to their controlling organizations. For these organizations and the IPs they control we also use supplemental information to understand which geographic points of presence are used. While our measurements do not rely on our ability to understand these associations, downstream analysis can benefit from them.

3.3.1 IP Metadata

We retrieve meta-data on resolved addresses to better understand what organizations they belong to, and whether two addresses are likely to be equivalent. The two signals we have found valuable to include in this process are the reverse PTR records for the addresses, and the WHOIS organization entry controlling the address. Reverse PTR records are contained in the 'in-addr.arpa.' pseudo-tld in the DNS hierarchy. They are maintained by the organizations controlling the IP address, and often provide

a canonical name when the IP belongs to a known service. The WHOIS database is a database of IP ownership maintained by IANA and its delegates that contains organizational responsibility, in the form of technical and abuse contacts, for addresses.

We perform direct lookups for both the PTR and WHOIS organizational contacts for all distinct IP addresses resolved. We then perform a clustering of each data set: All IPs with the same WHOIS organization are clustered into a WHOIS cluster, and all IPs with consistent PTR records are clustered together. To cluster PTR records, we use a simple heuristic: if all but the final dot-separated section of the returned records are equal, we put the IPs in the same cluster. For instance, a west coast resolution of `apple.com` has the PTR record of `a23-200-221-15.deploy.static.akamaitechnologies.com`, while an east coast resolver sees `a23-193-190-30.deploy.static.akamaitechnologies.com`. Since both cases end with `deploy.static.akamaitechnologies.com`, they are clustered together as part of the same entity.

3.3.2 Geolocation

During our collection and aggregation process we maintain a network, rather than geographical, view of the data. We prefer aggregation at a Class-C address level, which reduces calculations without losing precision or mixing IPs owned by different entities. Our other form of aggregation is on the AS level, to represent the aggregations of IPs which will see a similar view of the rest of the Internet. ASes represent business relationship which exist, and the AS which 'owns' an IP range is responsible for managing abuse and routing of packets for those IPs. As such, even when a sub-range is delegated, we assume the full AS experiences a consistent routing policy. There are always exceptions to such assumption: the Comcast AS contains clients on both of the east and west coast of the US, who will reach different data centers of many cloud services. Our use of AS aggregation will consider these results as a single combined data point. Likewise, the Google and Edgecast systems operate servers in many countries. When addresses in these ASes are used as resolvers, we consider them to be in the closest location to our measurement machine, the US.

For the visualization of infrastructure locations in the evaluation of this paper, we have to associate IPs with geographical locations. For this, we use three data sources: the country of registration for the *whois* point of contact (Used for AS location), the MaxMind [26] country-level database (Used for IP location), and the list of anycast prefixes from Cicalese et. al. [8]. When MaxMind geolocates different IPs within an AS to multiple countries, we use that list. Otherwise, we use the country of registration. Since MaxMind cannot handle geographic diversity

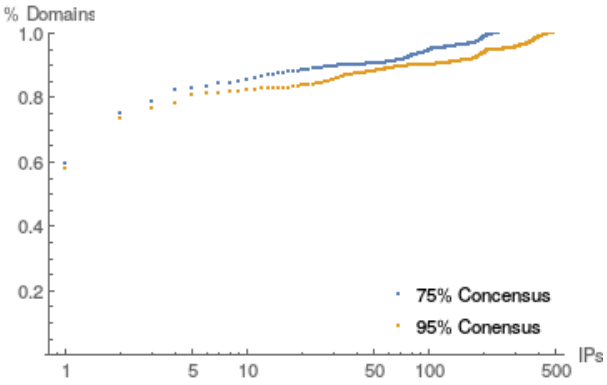


Figure 5: Number of IPs hosting different domains at two thresholds for dominant addresses. For 60% of domains, 1 IP accounts for 75% of all resolutions, and for 80% of domains, 10 IPs account for 95% of resolutions.

hidden by anycasting, we explicitly geolocate the points of presence of anycasting IPs, and use the closest point to a given resolver.

We find that 1% of distinct IPs resolved in a typical Satellite data match the anycast prefix list. To estimate the points of presence of these IPs, we measure latency from a range of vantage points, as in [22], resolving topology with [24]. In the future, we hope to learn these latencies through the DNS requests we already make using the technique in [17]. We find that since the CDNs we are identifying are highly distributed, we end up with observations which are either very small latencies indicating a point of presence near the vantage point, or are large enough to not impose additional constraints.

While these geographic heuristics are not infallible, they are largely accurate at the country level [32, 31]. As such, they provide a grounding for initial data exploration. When considering specific interference or deployment situations, it remains important to identify the relevant subsets of data. For instance, when we consider Iran in Figure 12, we manually limit our analysis to ASes of known ISPs in the country. We hope that a verifiable geographic database of infrastructure sourced from open measurements becomes available.

3.4 Aggregation

To support interactive exploration and analysis of collected data, satellite automatically aggregates the observed responses of each weekly collection. This automatic processing also materializes several views of the aggregated data which are used in subsequent analysis.

This automatic process attempts to parse each received packet as a DNS response, validates that it is a well-formed DNS response, and records the IP addresses returned. We tabulate these values for each resolver AS and domain. The resulting mapping is roughly 3 GB, and is used as the basis of subsequent processing. The 100-fold reduction comes from stripping the formatting and other

fields of DNS responses, and from aggregating responses by resolver AS. Scanning this file to calculate basic statistics takes under 5 minutes on a single 2.5GHZ core of our lab machines, and the format lends itself to parallel execution when more complex tasks are needed.

In addition to initial aggregation, we automatically build lookup tables for the set of IPs which have been resolved for each domain, and the total set of IPs seen as resolution answers. We also calculate the set of domains associated with each IP to facilitate reverse lookups of other domains potentially co-hosted on an IP. On a recent execution of Satellite, we saw a total of 5,337,315 distinct IPs resolved, located within 6,742 distinct ASes.

The domain resolutions we have collected already provide insight into the inner workings of popular websites. In Figure 5, we show how much diversity we found in the responses for each domain globally. If almost all responses return a single IP address, we can make the inference that the dominant IP is the canonical server for the domain. In other words, the domain is 'single homed'. In our monitored domains, we see this behavior in 60% of domain, the far left data points in the graph. Slightly further right in Figure 5 are domains which use simple load balancing schemes. We see that roughly 80% of domains have four or less 'dominant' IPs. This figure doesn't capture the use of anycast IP addresses, but does indicate that even for top domains, the majority have a single or small set of 'correct' addresses. The tail to the far right on the figure indicate domains which use geographically distributed infrastructure, and which require more complex analysis to determine whether individual resolutions are correct. For example, we record over 500 IP ranges for the `google.com` cluster, and over two hundred for many akamai hosted domains like `www.latimes.com`.

4 Evaluation

4.1 Address Validation

To validate our ranking and clustering algorithms, and our data collection process more generally, we make web requests to each resolved IP address as a potential location of each sampled domain. More specifically, we connect to each IP which has been seen as a candidate, and request the '/favicon.ico' file, using the domain as the 'Host' header. Slightly under half of the monitored domains have this file and can be validated in this way. We record hashes of all returned content, and compare these hashes against copies of the favicons fetched using local DNS resolution to determine whether an IP is correctly acting as a host for a given site.

Over a total of 965,522 completed resolutions, 82% of resolved IPs are deemed 'correct'. 5,479 domains are skipped in this validation, because no authoritative favicon is present, and validation is performed on the other 4,521. These domains are not used when we evaluate clustering performance.

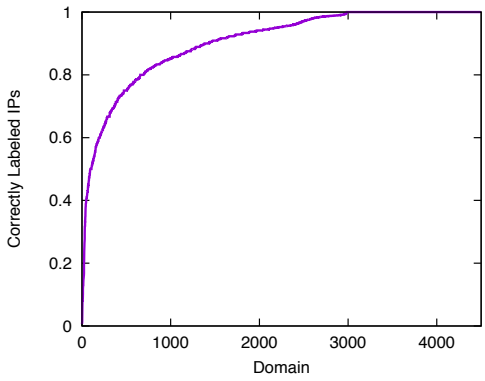


Figure 6: For each of the 4,500 domains with favicons, The fraction of distinct IPs resolved with a 'correct' `IPTrust` score over 0.5. Our automated classification matches favicon presence for over 90% of IP-domain pairs.

In Figure 6, we show the agreement between this validation process and the confidence scores for IPs used in our clustering algorithm. We treat an `IPTrust` score of 0.5 as trusted, but find similar results for other thresholds due to the polarization of trust scores by the iterative calculation. While there is noticeable divergence between the IPs in our scorings and the favicon results, over 95% of those failures are false-negatives (our algorithm was overly conservative in creation of clusters, and gives low scores to IPs the favicon process showed to be correct). The vast majority of these occur in situations where a single partition of IPs is normally resolved for a domain, but other IPs are also able to respond correctly when queried. Both Akamai and CloudFlare exhibit this behavior. Partial aggregation of these clusters has a minor effect on this view, since when domains are fully partitioned onto separate IPs we only consider our trust of those IPs we've actually seen resolved.

This validation technique is susceptible to manipulation by an adversary which returns the correct favicon image on an otherwise malicious server. We are not aware of any block pages behaving in this way.

In principle, validations like the use of favicons or signals like reverse DNS lookups can also be used in the clustering process to further refine which IPs are believed 'correct' for domains. To us though, this result shows that the DNS resolutions themselves are able to produce largely reliable mappings of CDN IP addresses.

We can also validate our clustering algorithms against the ground-truth of IP prefixes advertised by some CDN providers. For this validation, we consider the Fastly CDN, which uses a compact set of prefixes maintained at `https://api.fastly.com/public-ip-list`. We find that all 12 IP prefixes found by Satellite as the Fastly CDN cluster are included in the officially advertised list. The Satellite cluster con-

CDN	IP Space	Clustered ASes
CloudFlare	107008	75
Akamai	264960	489
Google	476416	1036
Cloudfront	128512	21
Incapsula	12288	17
Fastly	8192	17
Dyn	2304	9
Edgecast	24832	65
Automattic	3584	5
AliCloud	41728	42

Table 4: IPs in each of the ten largest shared infrastructure platforms. Variance in size between Dyn, Fastly, Automattic and the others is due to use of Anycast. Some ASes are significantly undercounted by clustering, Akamai has points of presence in over 1,000 ASes.

tains $72/80$ domains found using this ground truth list of IP prefixes. For geolocation, the MaxMind database reports multiple locations, accounting for 5 of the 10 Fastly countries, including the US, Australia, and three of four locations in Europe (mistaking Germany for France). The Australian class-c network prefix is identified as anycasting, which we resolve to 4 of the 5 additional locations – New Zealand, Japan, Hong Kong, and Singapore – agreeing with the results of [8]. These two techniques lead us to correctly find 8 of the 10 locations, missing Brazil and mistaking Germany for France.

4.2 Website Points of Presence

While we have shown in this paper that the Satellite technique is able to accurately map the IPs which are operated by targeted websites, we have not yet shown the implications of that data. Here, we attempt to characterize the dominant content distribution entities in the Internet today, and provide some insight into where they operate and the international nature of the Internet today.

In Table 4, we show the IP space we estimate for the largest CDN clusters. These platform each have unique network structures, and use a range of technologies including rotating IPs and anycast, which make it difficult to directly compare scale from these numbers. For instance, most Google IPs resolve to IPs within Google's own AS, while IPs from Akamai are largely resolved to IPs located in the ASes of consumer ISPs.

In Figure 7, we use the geolocation of ASes to count which countries these providers are located within. One striking feature of this geolocation exercise is to note that the 10 largest content distribution networks use IP addresses allocated to ASes registered in at least 145 countries. We trust MaxMind for these locations, but attempt to be conservative, including neither anycast resolution nor clustering the true extent of partitioned providers like Akamai. This undercounting is reflected in Table 4, which indicates the primary cluster we use for Akamai accounts for under half of the over 1,000 ASes they report [1].

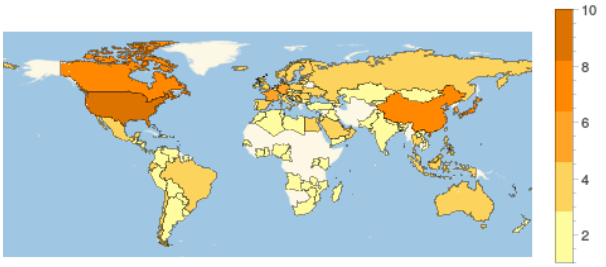


Figure 7: Points of presence of the CDNs from Table 3. Anycast is not included, indicating conservative counts.

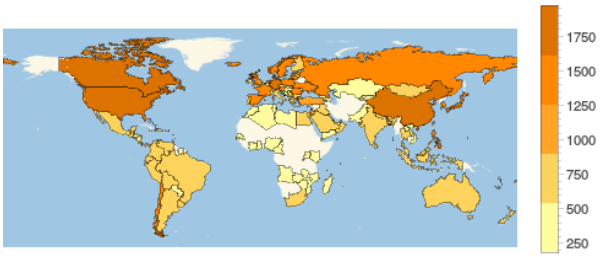


Figure 8: Number of sites resolved locally in each country.

In Figure 8, we plot how many domains are resolved within each country. We see at least 18% of all domains (2325) resolving to an in-country IP address for resolvers in China, while other countries like Mexico resolves only 5% (559) of domains locally. This view of domain locality can be used to understand which publishers have complied with local regulations, and to track how much Internet traffic will transit international links.

4.3 Interference

Our confidence scoring of how well IPs represent domains helps us address an ongoing pain point in interference measurement: how to know if a returned IP address is 'correct'. The primary issue in this determination traditionally has been whether an IP that is not the same as the canonical resolution is a CDN mirror or an incorrect response. Using CDN footprints along with more simple heuristics for single-homed domains allow us to identify instances of inaccessibility with higher confidence.

We measure interference through positive identification of the four categories in 2.4.2. These categories are conservative, but remain valid for not fully clustered CDNs.

Figure 9 shows the number of largely inaccessible domains found in a single snapshot of collected data. We find at least 5 of the monitored domains to be inaccessible in at least one Autonomous System in over 78 countries.

We then divide the instances of observed interference across other factors. Figure 10 shows a comparison of interference for sites on CDN infrastructure versus those which are single-homed. While roughly 80% of sites are single homed, we see as much interference is directed

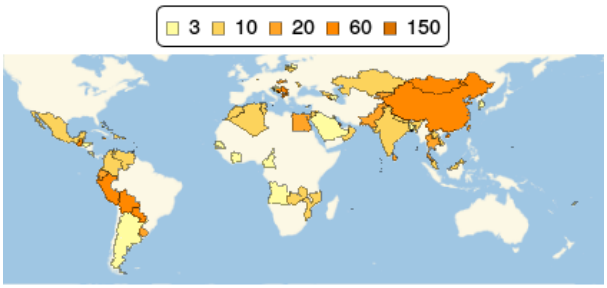


Figure 9: Number of domains inaccessible in each country.

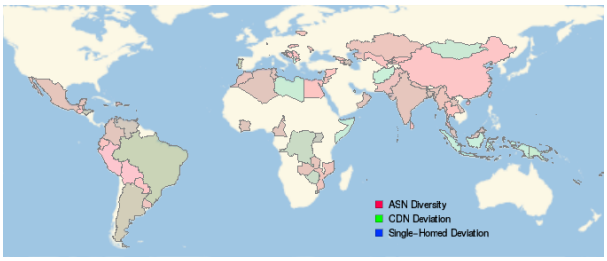


Figure 10: Types of interference by country. Anomalies are geographic, with some regions like China providing a diversity of false IP addresses, while others like Libya using a single block page. There are no occurrences of only 'CDN Deviation', or 'Single-Homed Deviation.' The relative shades indicate the mixture of the different categories present in each country.

at distributed sites, perhaps due to their popularity. This indicates that naive approaches have been missing a significant fraction of total interference instances.

It is possible for a censor to mask their interference from Satellite. Injecting DNS responses using a system of the type known to be in use by China could be targeted to miss an external observer, by only responding to requests originating within the Country or responding correctly to external queries. While much less visible to Satellite, these forms of interference would themselves be visible, and could even be less effective internally. The switch to other techniques like IP or keyword-based blocking would also not be visible in the current DNS data set.

4.4 Broader Implications

Our stated purpose in building Satellite and collecting data on the presence and accessibility of popular sites was to allow for new insights into the changing structure of the internet. What are those insights? Many of the implications are inextricably tied to real world events and politics, and reflect on the censorship practices and business environments of nation states. While we aren't comfortable claiming to understand these sociopolitical structures without accompanying real-world evidence, we can show value in the data in light of the larger trends occurring in Internet Governance.

In Figure 11 we show the delta of how many more domains are resolved within each country compared to six months prior, based on location of IPs with trust above 0.5

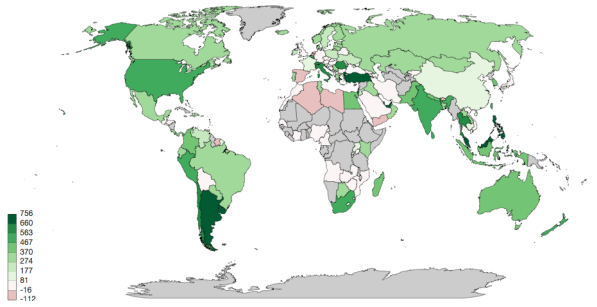


Figure 11: How many more sites resolve locally (to IPs within the country) in September 2015 compared to 6 months prior. This figure is based on a dataset of 8,800 domains which remained in the top 10,000 list at both sample points.

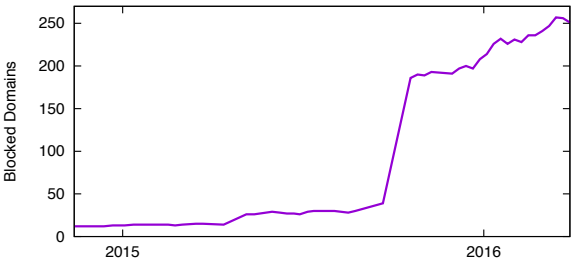


Figure 12: Number of domains detected to have anomalous resolutions in Iran since late 2014. An interactive version is at `http://satellite.cs.washington.edu/iran/`.

on a per-domain basis. What this shows for each country is how many new domains are now resolved internally where previously they would have been resolved to international servers. This shows the expansion of CDN infrastructure, but also an increasing ability of governments to regulate access within their national territories [10].

In Figure 12 we show the number of domains which are detected to have anomalous resolution across Iranian ISPs. We see a spike in the second half of 2015, which correlates with statements from the authorities there that they were beginning a second phase of filtering. More recently, Satellite has recorded additional inaccessible domains in the lead up to February 2016 elections.

5 Related Work

The active probing techniques used by Satellite build upon a long history of Internet measurement [24, 7]. The subsequent analysis of connectivity data has been tackled by previous generations of censorship measurement systems, though Satellite differs in the breadth of the measurements it aggregates and the way it handles noisy data.

Active scanning of the Internet has been used to measure important properties of ISPs, and has been shown to reasonably map individual CDNs [18, 6]. In particular, the rate of churn of DHCP reservations within consumer ISPs [27] has been estimated and the presence of Bluecoat DPI boxes [25] has been detected using active measure-

ment techniques. Active probing was used for the Internet census characterization of scale [3] and more generally in the web security space to measure the uptake of software updates and vulnerabilities [33, 12]. It has not yet to the best of our knowledge been used to independently measure the footprints of CDNs or longitudinal ISP-level interposition on traffic.

What to Measure: Determining domains of interest is by itself a tough problem. There are many billions of DNS records in use on the Internet [4, 14], and there are obvious deficiencies with the coverage or representativeness of lists of top sites. Previous measurement studies have used either top domains as reported by a neutral providers like Alexa [2], or more targeted lists they hand curate [15]. One of the most popular lists for censorship work is the list of sensitive domains maintained by Citizen Lab [30]. Satellite needs to measure a set of sites which reveal the shared infrastructure of CDNs, and we choose the Alexa top 10,000 domains as our base measurement set in order to achieve that coverage.

How to Measure: Researchers have invested considerable effort in the measurement of network interference, both by using participants within target networks [15, 20, 29] and through purely external mechanisms [9, 34]. DNS has been a measurement focus, largely because it is a commonly manipulated and unsecured protocol. DNS reflection against remote open resolvers has also been proposed for censorship measurement [36] as early as 2006. What we continue to lack is a system which is able to sustainably measure and act as a data repository for these measurements across both countries and time. Ripe Atlas [29] offers shared access to its distributed deployments of probes, but limits the types of measurements and rate-limits measurements such that regular probing of diverse domains by the deployment would require ownership of a significant fraction of the network.

Determining Site Presence: While determining which sites are of interest is hard, determining whether a given IP is a valid host for a site can be even harder. In their investigation of CDNs in 2008, Huang et. al [18] arrive at a similarly sized list of open resolvers as Satellite (280,000), and use them to map the Akamai CDN. They create their list of resolvers starting from DNS servers observed by Microsoft video clients, rather than direct probing. Specific CDNs like Google have also been crawled through the use of EDNS queries to simulate the presence of geographically diverse clients [6], but this is only possible for a small subset of deployments which support EDNS for redirection. Research focusing on censorship, like the analysis of ONI data [16], have used AS diversity to determine if IPs are valid for a domain, but do not explicitly consider CDN behavior.

There are also many commercial sites which offer traffic information for web sites. We know that some of this data is crowd-sourced through browser plugins, while other portions come from automatic robot crawling. For instance, the Alexa rankings are based off of a browser plugin which monitors the browsing habits of a small number of participating users. Some sites also show which sites run on identical IP addresses [19]. In practice we find that these systems appear to do direct lookups of IPs, since geographical distribution is not surfaced. They also do not appear to do significant identification of CDN IP spaces, since CDN'ed sites are not fully aggregated.

Determining Abnormal Behavior: Categorizing responses as normal or abnormal have typically been performed through the use of heuristics in how the response may deviate from expected behavior. This is true for both determining trust in a DNS response, and determining if a given connection is working as expected. These heuristics include metadata like the AS and reverse PTR record of the IP [16], behavior of HTTP queries to the server [21], and considering the aggregate prevalence of a given response [15]. More recent work has explored the use of aggregate statistical behavior to determine when network level behavior has changed [37]. These techniques provide valuable direction for Satellite, though there is not yet a comprehensive set of best practices for determining self-consistency and anomalies in our data set.

6 Conclusion

Satellite is already a valuable system for measurement of both CDNs and prevalence of interference. Our continued development efforts are focused on: (1) Improved reproducibility of geographic determination. (2) Developing an interactive visualization for interacting with data. (3) Integration of additional probing mechanisms for measurement of transport and IP level connectivity.

In this paper we have presented Satellite, a system for measuring web infrastructure deployments and availability from a single external vantage point. By lowering the bar for collecting, aggregating, and understanding we make this data much more accessible. Satellite the growing predominance of CDNs in the top Alexa domains. The same data shows evidence of growing interference of domain resolutions around the world. Satellite is a fully open platform, and both the data and code are available online at `satellite.cs.washington.edu`.

7 Acknowledgments

Special thanks to Sidney Berg, Adam Lerner, and the collaborators who have greatly improved Satellite. Thanks to Noa Zilberman for valuable insights in shepherding this paper to publication. This work is sponsored in part by the Open Technology Fund's Information Controls Fellowship Program, Google Research Award, and National Science Foundation (CNS-1318396 and CNS-1420703).

References

[1] Akamai. The akamai intelligent platform, 2016. akamai.com.

[2] Alexa, 1996. www.alexa.com.

[3] Anonymous. Internet census 2012, 2012. internetcensus2012.bitbucket.org.

[4] Anonymous. DNS census 2013, 2013. dnscensus2013.neocities.org.

[5] Anonymous. Towards a comprehensive picture of the great firewall's dns censorship. In *FOCI*. USENIX, 2014.

[6] M. Calder, X. Fan, Z. Hu, E. Katz-Bassett, J. Heidemann, and R. Govindan. Mapping the Expansion of Google's Serving Infrastructure. In *IMC*. ACM, 2013.

[7] B. Cheswick, H. Burch, and S. Branigan. Mapping and visualizing the internet. In *USENIX ATC*, 2000.

[8] D. Cicalese, D. Joumblatt, D. Rossi, M.-O. Buob, J. Augé, and T. Friedman. A fistful of pings: Accurate and lightweight anycast enumeration and geolocation. In *Computer Communications (INFOCOM)*, 2015.

[9] J. R. Crandall, D. Zinn, M. Byrd, E. Barr, and R. East. Conceptdoppler: A weather tracker for internet censorship. In *CCS*. ACM, 2007.

[10] R. Deibert and R. Rohozinski. Liberation vs. control: The future of cyberspace. *Journal of Democracy*, 2010.

[11] D. Dittrich, E. Kenneally, et al. The Menlo Report: Ethical principles guiding information and communication technology research. *US Department of Homeland Security*, 2011.

[12] Z. Durumeric, E. Wustrow, and J. A. Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security*, 2013.

[13] M. Elsner and W. Schudy. Bounding and comparing methods for correlation clustering beyond ILP. In *ILP-NLP*, pages 19–27, 2009.

[14] Farsight Security, Inc. Farsight DNSDB, 2010. dnsdb.info.

[15] A. Filastò and J. Appelbaum. OONI: Open observatory of network interference. In *FOCI*. USENIX, 2012.

[16] P. Gill, M. Crete-Nishihata, J. Dalek, S. Goldberg, A. Senft, and G. Wiseman. Characterizing web censorship worldwide: Another look at the opennet initiative data. *ACM Transactions on the Web*, 2015.

[17] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *IMC*. ACM, 2002.

[18] C. Huang, A. Wang, J. Li, and K. W. Ross. Measuring and evaluating large-scale cdns. In *IMC*. ACM, 2008.

[19] HypeStat, 2011. hypestat.com.

[20] ICLab, 2013. iclab.org.

[21] B. Jones, T.-W. Lee, N. Feamster, and P. Gill. Automated detection and fingerprinting of censorship block pages. In *IMC*. ACM, 2014.

[22] E. Katz-Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe. Towards ip geolocation using delay and topology measurements. In *SIGCOMM*. ACM, 2006.

[23] J. M. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. S. Tomkins. The web as a graph: measurements, models, and methods. In *Computing and combinatorics*. Springer, 1999.

[24] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI*. USENIX, 2006.

[25] M. Marquis-Boire, J. Dalek, S. McKune, M. Carrieri, M. Crete-Nishihata, R. Deibert, S. O. Khan, H. Noman, J. Scott-Railton, and G. Wiseman. Planet Blue Coat: Mapping global censorship and surveillance tools. 2013. citizenlab.org.

[26] MaxMind. Geoip, 2006. maxmind.com.

[27] G. Moreira Moura, C. Ganan, Q. Lone, P. Poursaied, H. Asghari, and M. Van Eeten. How dynamic is the ISPs address space? towards Internet-wide DHCP churn estimation. In *IFIP Networking*. IEEE, 2015.

[28] J. Muach. Open resolver project, 2013. openresolverproject.org.

[29] R. NCC. Ripe atlas, 2010. atlas.ripe.net.

[30] OpenNet initiative, 2011. opennet.net.

[31] I. Poese, S. Uhlig, M. A. Kaafar, B. Donnet, and B. Gueye. Ip geolocation databases: Unreliable? In *SIGCOMM CCR*. ACM, 2011.

[32] Y. Shavitt and N. Zilberman. A geolocation databases study. *Selected Areas in Communications, IEEE Journal on*, 2011.

[33] Shodan. shodan, 2013. `shodan.io`.

[34] J.-P. Verkamp and M. Gupta. Inferring mechanics of web censorship around the world. In *FOCI*. USENIX, 2012.

[35] P. Vixie. Extension mechanisms for DNS (EDNS0). RFC 2671, 1999.

[36] S. Wolfgarten. Investigating large-scale Internet content filtering. Master's thesis, Dublin City University, Ireland, 2006.

[37] J. Wright, A. Darer, and O. Farnan. Detecting internet filtering from geographic time series. *arXiv preprint arXiv:1507.05819*, 2015.

[38] Y. Xie, F. Yu, K. Achan, E. Gillum, M. Goldszmidt, and T. Wobber. How dynamic are ip addresses? In *SIGCOMM CCR*. ACM, 2007.

Subversive-C: Abusing and Protecting Dynamic Message Dispatch

Julian Lettner[*] *Benjamin Kollenda*[†] *Andrei Homescu*[§] *Per Larsen*[*§] *Felix Schuster*[¶]
Lucas Davi[‡] *Ahmad-Reza Sadeghi*[‡] *Thorsten Holz*[†] *Michael Franz*[*]

[*]*UC Irvine* [†]*Ruhr-Universität Bochum*
[§]*Immunant, Inc.* [¶]*Microsoft Research*
[‡]*Technische Universität Darmstadt*

Abstract

The lower layers in the modern computing infrastructure are written in languages threatened by exploitation of memory management errors. Recently deployed exploit mitigations such as control-flow integrity (CFI) can prevent traditional return-oriented programming (ROP) exploits but are much less effective against newer techniques such as Counterfeit Object-Oriented Programming (COOP) that execute a chain of C++ virtual methods. Since these methods are valid control-flow targets, COOP attacks are hard to distinguish from benign computations. Code randomization is likewise ineffective against COOP. Until now, however, COOP attacks have been limited to vulnerable C++ applications which makes it unclear whether COOP is as general and portable a threat as ROP.

This paper demonstrates the first COOP-style exploit for Objective-C, the predominant programming language on Apple's OS X and iOS platforms. We also retrofit the Objective-C runtime with the first practical and efficient defense against our novel attack. Our defense is able to protect complex, real-world software such as iTunes without recompilation. Our performance experiments show that the overhead of our defense is low in practice.

1 Introduction

The primary programming environment on Apple's OS X and iOS platforms uses a language called Objective-C, which extends the C language with object-oriented constructs. Many of the main application programs on Apple's platforms, such as Safari, iTunes, etc. are built using Objective-C, which differs from C++ in the way that dynamic dispatch of function calls is implemented. In spite of its importance to commercial software platforms, it has attracted little scrutiny from systems security researchers.

The latest code-reuse mitigation being deployed—CFI—makes traditional ROP [30] attacks harder to construct. CFI computes an approximation of an application's control-flow graph (CFG) and verifies that all indirect branches follow valid CFG edges at run time [1]. In contrast to randomization-based defenses [26], CFI is secretless and cannot be bypassed via information leakage. Like other mitigations, CFI must trade off security (precision) for performance. Coarse-grained CFI policies [43, 44] leave a small fraction of code locations available for reuse by adversaries—enough to mount ROP attacks [16, 22, 32]. The deficiencies of coarse-grained CFI renewed interest in more precise policies. Devising such CFI policies typically requires source code access, because structural information required to compute a complete and precise CFG is lost during compilation. The recent COOP [31] code-reuse technique exploits the imprecision of non-C++ aware CFI implementations on Windows and Linux. Specifically, the attacker manipulates the *virtual method tables (vtables)* of C++ objects in memory such that a sequence of attacker-chosen regular virtual methods is executed via likewise regular virtual method call sites. Unlike ROP, COOP does not violate the integrity of return addresses or produce corrupted call stacks and therefore remains undetected by generic CFI policies [17, 20]. Moreover, the high-level structure of C++ code (e. g., class hierarchy and dynamic object types) cannot be fully recovered without source code, so malicious COOP control flows are difficult to distinguish from benign ones even for *C++-aware* CFI policies computed by binary analysis. In terms of expressiveness and flexibility, COOP is comparable to ROP in C++ environments [14, 31]. Still, it remains unclear whether COOP is limited to C++ code on Windows and Linux or whether it is a generic threat on par with ROP.

This paper shows that programs written in Objective-C suffer from a systematic vulnerability that enables COOP-style exploits against Objective-C on OS X and iOS. Like C++, Objective-C extends the C programming language with object-oriented constructs. Although both languages add *dynamic dispatch* of function calls to C, the implementation of this feature differs greatly between C++ and Objective-C. Whereas C++ fixes the vtable for each class

at compile time, Objective-C enables full *late binding* by (re)mapping literal method names to actual functions dynamically at run time. We dub our new class of attacks Subversive-C and demonstrate its viability against applications using AppKit, a commonly used framework on Mac OS X, by constructing a proof-of-concept exploit.

We also show how Subversive-C exploits can be mitigated. Our mitigation strategy can be retrofitted onto existing systems without requiring recompilation of the programs being protected and has very little overhead.

An important insight is that in many cases, an attacker can use COOP, Subversive-C, or a combination of both, because non-trivial OS X and iOS applications like Safari or MS Office typically contain both Objective-C and C++ (standard libraries or ported code from other platforms) components. In fact, it is even valid (and common) to tightly interweave Objective-C and C++ semantics. Such "Objective-C++" code is accepted by the GCC and Clang compilers. Hence, effective code-reuse defenses for OS X and iOS need not only to consider high-level semantics of Objective-C, but also those of C++.

In summary, our main contributions are as follows:

- **Novel Offensive Technique** We present Subversive-C, a new offensive technique that reuses entire Objective-C methods by carefully arranging the metadata used to dispatch messages in the Objective-C runtime. The dynamic nature of Objective-C coupled with whole-function reuse renders existing integrity and randomization-based defenses ineffective against Subversive-C exploits.
- **Hardened Objective-C Runtime** Because existing defenses cannot protect against Subversive-C with low overheads, we developed a new defensive technique to prevent adversaries from manipulating and corrupting metadata used by the Objective-C runtime. Specifically, we retrofit the Objective-C runtime with integrity checks in the lookup processes that handle Objective-C message dispatch. Our hardened runtime is fully compatible with the runtime shipped with OS X and can protect complex, real-world applications such as iTunes.
- **Realistic and Extensive Evaluation** We demonstrate a fully-fledged Subversive-C attack targeting the AppKit library. We also provide a careful and detailed evaluation of our hardened Objective-C runtime. We report a 1.54 % aggregate overhead for complex, real-world applications.

2 Technical Background

In the following, we provide a brief overview of the technical concepts we use in the rest of this paper. We discuss dynamic message dispatch in Objective-C and present an overview of resarch on code-reuse attacks with a specific focus on COOP.

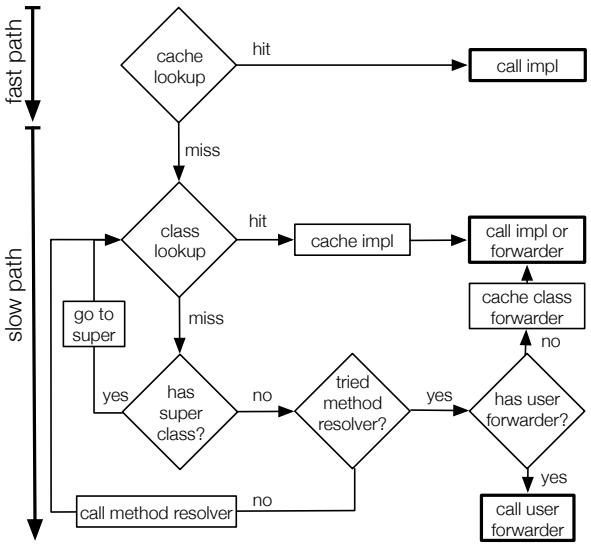


Figure 1: Fast and slow paths when dispatching messages.

2.1 Dynamic Message Dispatch

Objective-C is an object-oriented programming language that extends the C language with dynamically dispatched Smalltalk-style messaging. Where C++ programmers invoke (virtual) methods of objects, Objective-C programmers send messages to objects. Each message has three components: i) the **receiver** object; ii) the **selector**, an identifier naming the method that receives the message; and iii) zero or more **arguments**.

Although Objective-C is a statically compiled language, the targets of message dispatches are resolved at run time. At every message dispatch location, the compiler simply emits a call to the `msgSend` function (or one of its variants) in the Objective-C runtime. The purpose of the `msgSend` function is to locate the appropriate method for a given (receiver, selector) pair and subsequently execute it.

Figure 1 illustrates the message dispatch algorithm as implemented in Apple's Objective-C runtime. It consists of a fast path and a slow path. The slow path retrieves the method implementation corresponding to a given selector by searching through all methods defined by the class of the receiver object and all its ancestors. The search operates on compiler-generated metadata attached to each object as shown in Figure 2.

The lookup algorithm starts with the class of the receiver and checks the selector against all methods defined by the receiver's class. If no method is found, the methods of the parent class is searched until a method implementation is found or the root of the class hierarchy is reached. If neither the class itself nor any of its ancestors contain a method implementation, the runtime allows the class to

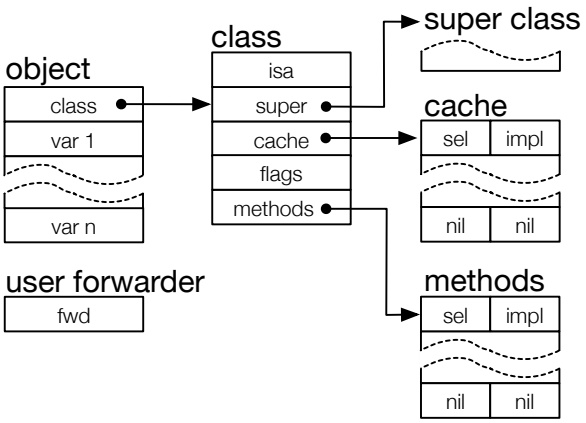


Figure 2: Layout of objects, classes, and lookup caches.

dynamically add an implementation for the given selector. If the class provides a "method resolver" function, the runtime calls it. The resolver (depending on its implementation) may add a new method to the class. The runtime then repeats the entire lookup process, in case the method added by the resolver corresponds to the input selector.

The fast path speeds up message dispatch by storing the results of each lookup in a per-class cache, and reuses previous results if available. At the beginning of each message dispatch, the runtime queries this cache for a method pointer as shown near the top of Figure 1.

The cache entries are stored in memory as a linear array of (selector, method pointer) pairs. The class metadata includes a pointer to its corresponding cache. The runtime performs the lookup using a linear probing algorithm. The lookup starts from a location computed by hashing the selector itself and proceeds linearly through the array until either a match is found—a *cache hit*—or an empty entry is reached—a *cache miss*.

In case neither the cache lookup nor the slow class hierarchy walk find the method for a selector, the runtime performs one last step before exiting with an error. If the class provides a "forwarder" function, the runtime calls this function with the selector, allowing classes to dynamically respond to new messages at run time or forward messages to other objects. Additionally, the Objective-C runtime allows the application to install a "user forwarder" that overrides all per-class forwarders. If this handler is installed, the runtime calls it at the end of the method lookup process. The handler pointer is stored in a writable global variable, which can be manipulated by adversaries.

2.2 Exploitation and Code-Reuse

C/C++/Objective-C eschew memory and type safety features of modern languages and require manual memory allocation and deallocation. This leads to a steady stream of memory management errors.[1] Attackers exploit the presence of these errors to craft malicious inputs that hijack the control flow of the application. The classic stack smashing attack injects code and redirects execution to it by overwriting the return address stored past the end of an overflowed buffer [2]. Thanks to modern mitigations such as data execution prevention (DEP), which disallows memory regions that are both writable and executable, code injection is all but obsolete. Therefore, modern exploits reuse legitimate code to bypass DEP. There are many known variants of code-reuse attacks. The main differences are the granularity at which legitimate code is reused. ROP reuses short instruction sequences ending in returns called gadgets [30,34]. Other variants reuse whole functions, including Return-into-`libc` (RILC), COOP, and our novel Subversive-C technique. Another key difference is the dispatching mechanism used to transfer control between the code snippets being reused. ROP and RILC use return instructions or indirect jump/call instructions [11]. COOP-style attacks use special "main-loop gadgets" to iteratively or recursively dispatch a sequence of method calls controlled by a malicious payload.

An important prerequisite of a code-reuse attack is knowledge of the target's memory layout, because the payload in a code-reuse attack necessarily (directly or indirectly) references existing code locations. Thus, address space layout randomization (ASLR) complicates code-reuse attacks because it randomizes the base address of linked libraries at load time. However, this is only a small hurdle for a practical code-reuse attack since information leakage or memory disclosure attacks often enable attackers to undermine ASLR [33,35,36,38].

2.3 Counterfeit Object-oriented Programming

COOP is a code-reuse technique targeting C++ software [31]. In COOP, a sequence of attacker-chosen C++ virtual methods (also called *vfgadgets*) is executed on attacker-injected objects (also called *counterfeit objects*). Each vfgadget in such a sequence fulfills a specific task, such as reading a value into a register and may have certain side-effects. Executed one after another, the vfgadgets implement the malicious functionality desired by the attacker, e. g., the execution of a shell command. Put simply, (short) virtual methods are to COOP attacks what gadgets are to ROP attacks. Whereas a ROP attack is initiated by injecting a "fake stack" (containing fake return addresses) into the target address space, a COOP attack injects a collection of "counterfeit objects", typically using a single attacker-controlled write. Each counterfeit

[1] For new code, disciplined use of modern coding techniques like smart pointers for C++ and automatic reference counting (ARC) for Objective-C alleviate this problem.

object corresponds to exactly one vfgadget and carries a corresponding pointer to a vtable.[2]

In ROP and related techniques, data primarily flows through registers and the stack from one gadget to another. In contrast, data may flow in three different ways between COOP vfgadgets: *(i)* through method arguments, *(ii)* through global variables, and *(iii)* through member fields of *overlapping* counterfeit objects. The latter is a pattern specific to COOP, which can greatly facilitate the creation of meaningful vfgadget chains. For example, vfgadget *1* may read a value from memory and store it in the field *x* of counterfeit object *A* and vfgadget *2* may increment field *y* of counterfeit object *B*. By making objects *A* and *B* overlap such that *A.x* and *B.y* map to the same address, the methods *1* and *2* can be used in conjunction to read and increment a value.

Different techniques for chaining the execution of vfgadgets in a COOP attack have been described in previous work [14, 31]. Using one of these techniques, the attacker initially corrupts a C++ object used by the target application such that a subsequent virtual method call is maliciously diverted to a central dispatcher vfgadget. (In a ROP attack, the control flow would instead be diverted to the first gadget, which usually pivots the stack pointer.) In the simplest case, a so called *main loop* (ML-G) vfgadget is used. Briefly, an ML-G is a virtual method that invokes virtual methods on a collection of C++ objects. By making an ML-G operate on a collection of counterfeit objects, the chained execution of vfgadgets becomes possible. An example ML-G is discussed in Section 4.2.

3 Threat Model and Assumptions

The assumptions underpinning this research are:

- **Data** The attacker can arbitrarily read and write data pages as allowed by the page permissions. Specifically, the internal data structures of the Objective-C runtime can be read, written, and corrupted. However, we assume that ASLR is in place to randomize the locations of program and runtime data structures.
- **Code** We assume that DEP prevents code injection by disallowing the execution of writable pages.
- **Runtime** We assume that the runtime is protected using fine-grained code randomization [26], as well as an implementation of execute-only memory (XoM), such as XnR [6], Readactor [13], or HideM [21], that prevents attackers from using information leaks to retrieve the code of the runtime. We also rely on these defenses to secure secret keys (see Section 5.4).
- **Control flow** Since Objective-C is a superset of C, we assume the C parts of the application and runtime are protected using appropriate mitigations (CFI, randomization, or equivalent defenses). Defenses such as Mobile CFI (MoCFI) [15] can be used to protect Objective-C code from control-flow hijacking.

Note that these assumptions are realistic and match the capabilities of a real-world attacker. They also match the adversarial models used in closely related work [13, 14].

4 Subversive-C

In this section, we demonstrate that the principles of the COOP attack are not only applicable to C++ but also to Objective-C. Conceptually, a Subversive-C attack proceeds analogously to a COOP attack: the attacker diverts an application's control flow such that a sequence of attacker-chosen Objective-C methods (vfgadgets) is executed on injected counterfeit objects. The first method executed in such a sequence is necessarily a dispatcher vfgadget, e. g., a *main loop vfgadget* (ML-G) as described in Section 2.3. COOP and Subversive-C are closely related in the way they rely on counterfeit objects and vfgadgets. However, as they target different programming languages, COOP and Subversive-C counterfeit objects differ in their layouts. For COOP it is sufficient to create objects that reference a vtable, whereas the Objective-C runtime features a more involved class layout. Therefore, an attacker must forge multiple data structures to launch a Subversive-C attack. The exact procedure is described next in Section 4.1. Section 4.2 then presents a concrete Subversive-C attack against applications that use the AppKit library.

For brevity, we limit the discussions in this section to Apple's OS X operating system and the x86-64 architecture. However, all techniques and concepts extend to Objective-C code running on iOS and ARM.

4.1 Exploiting the Objective-C Message Dispatch Mechanism

The Objective-C runtime implements two different ways (*slow* and *fast*, see Section 2.1) to resolve a class-selector pair to a function address. We now describe how the attacker can exploit the Objective-C runtime's slow path and fast path lookup mechanisms in order to control the methods invoked on counterfeit objects in a Subversive-C attack. These techniques are specific to Subversive-C and are the key differentiators with respect to COOP.

Slow Path As described in Section 2.1, when a cache lookup for a selector fails, the `msgSend` function does a slow search through all methods available for the receiver object. The corresponding data structures are partly stored in read-only memory and cannot be modified by the attacker at run time. Hence, in order to freely choose the

[2] In C++, every object of a class with virtual methods carries a pointer to the class's fixed vtable. Whenever a virtual method is to be executed on a C++ object at run time, this pointer is dereferenced and the respective method's address is fetched from the table.

vfgadgets executed in a Subversive-C attack, the attacker needs to inject new fake data structures alongside each counterfeit object. Concretely, each counterfeit object needs to reference its own fake *class struct*[3] which in turn references its own fake *method list* (cf. Figure 2).

Each entry in a class's method list links a function pointer to a selector. It is thus sufficient to inject fake method lists with a single entry. This entry must link the fixed selector used in the dispatcher gadget to the vfgadget that is to be executed on the corresponding counterfeit object. In turn, the injected fake class struct must be shaped in such a way that `msgSend` actually takes the slow path and evaluates the given method list as desired. A straightforward way to ensure this is to null-out the cache-related fields in the class struct (i.e., invalidate the cache) and to mark the class as *initialized* by setting the corresponding bit in the `flags` field (not shown in Figure 2).

Instead of creating valid class structs from scratch, for increased stealthiness and simplicity, an existing class struct that is compatible with the given dispatcher can be copied and modified as needed.

Fast Path Instead of invalidating the cache of counterfeit objects, the attacker can also opt to exploit the fast path look-up by injecting fake class structs with *valid* cache entries linking the dispatcher's selector to vfgadgets. Doing so is simple, as the caching mechanism does not use a secure hashing function and, in any case, its parameters can also be directly rewritten by the attacker. Hence, the attacker can arbitrarily precompute valid cache entries offline and incorporate them into fake class structs.

Forward Handlers In addition to forging method lists and caches, a third option for the attacker to execute arbitrary methods from a dispatcher is to abuse *forwarders*, which are introduced in Section 2.1: existing forwarders structs (cf. Figure 2) could be manipulated or fake ones could be injected such that vfgadgets are executed instead of actual forwarder handlers. In this approach, the attacker needs to make sure that both the slow and the fast path fail for all counterfeit objects for the given dispatcher—e.g., by injecting fake class structs with an invalid cache and an empty method list.

4.2 Proof-of-Concept Exploit

To demonstrate the general applicability of our technique, we constructed a Subversive-C attack for the x86-64 version of the AppKit library. AppKit is part of the Cocoa framework which encompasses Foundation, AppKit and Core Data. AppKit in particular is used to create graphical user interfaces. As such it is included in most graphical Objective-C programs, including iTunes, Safari, Pages, Keynote, and many other widely used applications from Apple and third parties. The Objective-C methods used in the attack are given in Table 1. We extended the framework that Schuster et al. [31] used to create the COOP chains to account for the differences between C++ and Objective-C. The framework uses the SMT solver Z3 [18] to construct a buffer with the constraints defined by the layout of the objects and their required relative offsets to each other. (Recall that typically at least some counterfeit objects overlap.)

For our proof-of-concept exploit, we require a program that contains a memory corruption vulnerability allowing an attacker to place data in the target process as well as overwrite a pointer to an Objective-C instance used during execution. To reliably bypass ASLR, we further require an information leak to disclose the position of the data injected and the location of the instance pointer we override with our own counterfeit object. Since our gadgets are sourced from the AppKit library, this library must also be loaded by the target process. We simulate a suitable vulnerable application by creating an Objective-C program that requires the AppKit library and lets us inject attacker-controlled data in the address space. This data is then interpreted as an Objective-C object, more precisely as our *initial object*, which will start our chain. After this first dispatch the execution is driven entirely by our counterfeit objects.

High Level Overview For our proof of concept we opted to construct a chain that leads to the use of an *invoke gadget* to call an arbitrary function, in this case we chose `system()`. The other gadgets are used to prepare the call by calculating the function address based on import address table (IAT) entries and arranging arguments in memory correctly. After injecting the counterfeit objects into the target process, the chain is started by dispatching a message on the *initial object*, which directs the control flow to the *main loop gadget*. This ML-G dispatches calls to all other gadgets that perform the necessary computations. The chain reads the address of `libsystem!strlen()` from the IAT and adds a precomputed offset to it. The result is then used as the target for the *invocation gadget* (INV-G in COOP parlance [31]). The argument for this call is also located in the attacker-controlled memory and is passed as well. Any precomputed data is passed via fields in the injected counterfeit objects. In Objective-C, an object's fields are also referred to as its *instance variables*.

Initial Object The initial object is the first counterfeit object and is not part of the actual chain. It is designed such that dispatching the corresponding selector on it will enter the ML-G instead of the intended function. Additionally we pass required arguments, in this case the address of the gadgets, as instance variables.

Main Loop At the core of our attack lies the main loop gadget. We use an array-based ML-G (entry 1 in table 1)

[3] In practice, the class struct is oftentimes split into a read/write and a read-only part by the compiler. For brevity, we do not make a distinction between the two here and consider them as one coherent data structure.

#	Method name (AppKit)	Type	Description
1	`[NSTextReplacementNode dealloc]()`	ML-G	main loop
2	`[NSUndoTextOperation affectedRange]()`	LOAD-R64-G	load `rdx` from instance var.
3	`[NSPersistentUIRecord setEncryptionKey:](uint8_t[16])`	R-G	load `rdx` from address `rdx+8`
4	`[NSPanelController stringValue]()`	LOAD-R64-G	load `rcx` from instance var.
5	`[NSMatrix cellAtRow:column:](int64_t, int64_t)`	ARITH-G	`rdx = rdx·[self+0xf8] + rcx`
6	`[NSScrollingScoreKeeper setHoldCount:](int64_t)`	W-G	write `rdx` to instance var.
7	`[NSCustomReleaseData dealloc]()`	INV-G	invoke instance var. as function ptr.

Table 1: Our Subversive-C chain in the standard OS X AppKit library (x86-64) calculates the address of `system()` in `libsystem` and invokes `system("/bin/sh")`. Gadget type names are according to previous work [31].

which iterates over an array of objects and dispatches a constant selector on every entry. Each counterfeit object is an entry in this array. The pseudo code representation of our ML-G is shown in Listing 1; line 5 invokes the selector `release` on every counterfeit object in the injected array. While this particular ML-G is limited to 28 entries in a single array, inserting the ML-G itself again as the 28th entry allows the chaining of more gadgets.

Listing 1: ML-G in NSTextReplacementNode dealloc.

```
1 children = self->children;
2 counter = 0;
3 while (children[counter] != NULL
4     && counter < 28) {
5   [children[counter] release];
6   counter++;}
```

Read Gadget We use two read gadgets (#2 and #4) to load `rcx` and `rdx` from instance variables. As these are *argument registers*, they are guaranteed to remain unaltered by `msgSend`. We load `rdx` with the address of the IAT entry of `strlen()` and `rcx` with the offset between `strlen()` and `system()` in `libsystem`.

Read Gadget with Dereference As we only assume the address of the AppKit module to be given, the address of `system()` in `libsystem` needs to be calculated dynamically. To this end, we read a pointer to `libsystem` from the IAT of the AppKit module and, in the next step, add a constant offset to it. The gadget we use (entry 3) loads `rdx` with the 64-bit value pointed to by `rdx+8`. As we control the value of `rdx` with gadget #2, we can read from a chosen address here. We use this to load `rdx` with the address of `strlen()` from AppKit's IAT.

Arithmetic Gadget At this point `rdx` and `rcx` contain attacker-controlled values and can be used to calculate the actual address of `system()`. Gadget #5 adds both registers and stores the result to `rdx`.

Store Gadget Due to the semantics of our *invocation gadget* (INV-G) (see next step) we need to store the calculated address of `system()` in a specific instance variable of counterfeit object #7. Thus, the two counterfeit objects corresponding to gadgets #6 and #7 need to overlap: gadget #6 stores the function pointer in `rdx` in an instance variable of its counterfeit object; gadget #7 reads this pointer from an instance variable in its counterfeit object (which maps to the same address) and invokes it.

Invocation Gadget The original purpose of our INV-G (#7) is the invocation of a custom deallocator specified via an instance variable. The argument that is passed is also read from an instance variable. This means we both control the function called and its argument. Here, we use this to execute `system("/bin/sh")`.

5 Mitigating Subversive-C

A key insight of our attack is that it targets data structures specific to the Objective-C runtime, much like COOP targets the C++ specific vtable. Therefore, we build our defense around protecting the integrity of these data structures. Unlike C++ vtables, the data structures used by `msgSend` are mutable which means COOP defenses such as vtable randomization [14] are not suitable to protect the Objective-C runtime against Subversive-C. Instead, we choose to base our defense on message authentication to detect malicious tampering.

We add a message authentication code (MAC) to every sensitive field or data structure in the runtime as shown in Figure 3, and use this MAC to verify the integrity of the data structures before sensitive control flow transfers, i.e., those that indirectly use the contents of the data structures. Every time the runtime changes the contents of one of its structures, it also updates the MAC. Thus, an attacker can no longer alter data structures without needing to update the associated MAC. However, each MAC computation has two inputs: the message (data) to authenticate and a secret key. Without both inputs, a correct MAC cannot be computed. Knowing the secret keys would allow attackers to tamper with runtime data structures, so we store them in a key store which attackers cannot read. We describe the key store in detail in Section 5.4.

In the following, we first describe our different approaches to the stages of method lookup, as they have different requirements (most notably the tolerable overhead). Subsequently, we explain the implementation of our secure key store which protects keys from attackers.

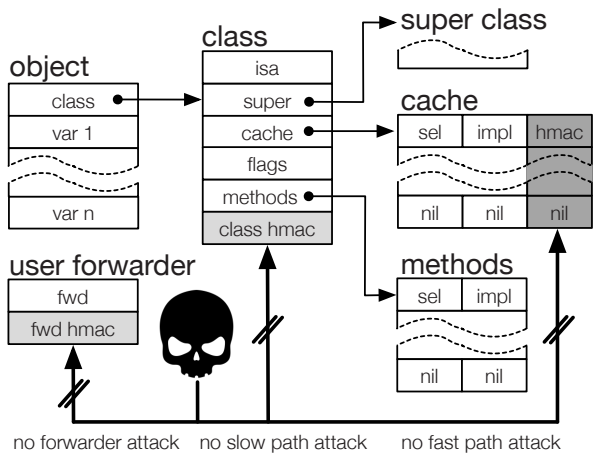


Figure 3: HMACs are used to ensure the integrity of class metadata and caches.

5.1 Securing the Slow Path

To protect the slow path lookup, we repurpose an unused field in the `class` structure to store a MAC (or more precisely a HMAC) as depicted in Figure 3. The hash is populated during class initialization and checked before any class metadata is used for method lookup. If a discrepancy is detected, program execution is aborted with an error message. To compute the hash, we chose the HMAC-MD5 function with the following inputs:

- The **method list** consisting of flags, entry count and an array of `method` structures.
- The **superclass** field to prevent the attacker from modifying the class hierarchy.
- The **flags** field to prevent the attacker from removing the *initialized* bit. An unset *initialized* bit forces the runtime to rebuild the method list (a process which the attacker could tamper with).
- The **isa** field which points to the meta-class of the current class.
- The **address** of the `class` object to uniquely identify the class. A unique identifier is needed to distinguish between similar classes, such as siblings, preventing the attacker from copying the method lists and hash values between them (in such cases, the superclass pointer and flags match).
- A **secret key**—K_{class}—retrieved from a secure key store, which we discuss in more detail in Section 5.4. We use a single global K_{class} for all classes in the application.

Let X be the concatenation of all the elements in the above list except the secret key. We use the following HMAC:

$$H_{\text{class}}(X, K_{\text{class}}) = \text{HMAC}_{\text{MD5}}(X, K_{\text{class}}) \qquad (1)$$

Our choice of the HMAC function is a pragmatic one: HMAC-MD5 is relatively fast, still considered secure [7] (in contrast to MD5), and is available through a library already linked by the Objective-C runtime. Note that the choice of HMAC is a security parameter in our defense; we can replace HMAC-MD5 with any stronger (but likely also slower) MAC in case attacks against HMAC-MD5 appear.

The core assumption of our protection scheme is that the attacker does not know the secret key and hence cannot modify the method list or other metadata used during method lookup without being detected. However, metadata may also change for legitimate reasons. Objective-C is a dynamic language which provides APIs for, e.g., adding classes and methods at run time. We support legitimate changes to metadata via provided APIs by making the change, followed by recomputing the HMAC field.

Note that computing the MAC adds considerable overhead to the slow path lookup (see Section 6.2 for empirical evaluation results). However, lookup results are cached so the slow path is only executed once per (class, selector) pair. Therefore, the steady state program performance remains unaffected. This is also reflected in the implementation of the runtime: the fast path consists of highly-optimized assembly code while the slow path is simply written in C.

5.2 Securing the Fast Path

We protect the fast path in a manner similar to the way we secure the slow path. We implement an authentication mechanism for cache entries that detects any tampering. However, in our practical experiments, we have encountered applications, e.g., iTunes, that modify cache entries directly, i.e., writing to the entries in memory instead of using the runtime API, in much the same way an attacker could tamper with the cache. Therefore, we must allow changes to the cache originating outside the runtime and make sure we detect them and fall back to the slow path.

We implement this by extending the fast path lookup algorithm with an additional authentication step for *cache hits*, as shown in Figure 4. This additional step computes the MAC of the cache entry and checks it against the MAC stored inside the entry. If the hash matches, the algorithm continues normally. Otherwise, the algorithm considers the authentication failure as a *cache miss*. We also modified the runtime to update the stored MAC on changes to a cache entry.

Each cache entry contains two pointers: the selector and the method implementation pointer, as shown in Figure 3. Using these pointers as the MAC input ensures that the attacker cannot modify existing cache entries or add new ones. However, the attacker could still copy entries between caches for different classes, and we wouldn't be able to detect this. Therefore, we add a third pointer to the MAC input: the pointer to the class that owns the

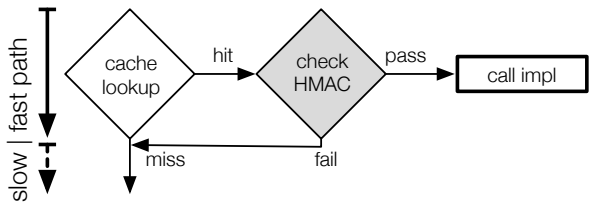


Figure 4: Fast path secured with MAC integrity check.

cache. This prevents the attacker from copying valid cache entries between classes, as each cache entry is now associated with a class.

Unlike the slow path, performance is critical on the fast path and every additional instruction can have a significant impact. Therefore we selected a MAC that we can implement in as few assembly instructions as possible, and easily integrate into the existing cache lookup algorithm. The NH hash function family used in UMAC [8] meets our performance requirements, so we use a modified version of NH as part of our MAC:

$$H_1(X,K) = \sum_{i=0}^{i\leq 2} (X_L[i] + K_L[i]) * (X_H[i] + K_H[i]) \quad (2)$$

where:

- $X = (class, sel, imp)$ is the 192-bit concatenation of the three pointers to hash: the class pointer, the selector and the method implementation.
- $K = (K_0, K_1, K_2)$ is the 192-bit secret key retrieved from the key store.
- $X_L[i]$, $K_L[i]$, $X_H[i]$, and $K_H[i]$ are the low and high 32-bit words of the i^{th} element of X and K, respectively.

The H_1 function has very low collision probability, but is vulnerable to known plaintext attacks (given a large enough sample of $H_1(X,K)$ outputs and their corresponding X inputs, the attacker can compute the K), and therefore insufficient to use as a MAC. UMAC strengthens NH against these attacks by XORing its result with a random number produced by applying a pseudo-random function (PRF) to a nonce.

Using a strong PRF in this case would take too many processor cycles, however, so we use a faster alternative in the form of a fixed-size table T of random 64-bit numbers. We generate this table at program load time, and store it in memory securely as described in Section 5.4. To compute the 64-bit HMAC of a cache entry, we compute H_1, use its output to index into T, and use the resulting value as the output. To simplify indexing, we always allocate T as a table of size $|T| = 2^N$. To compute the index we truncate the output of H_1 to 32 bits and then use the highest N bits. The final form of the HMAC becomes:

$$H_{\text{cache}}(X, K_{\text{cache}}) = T[H_{1[31:(31-N)]}(X, K_{\text{cache}})] \quad (3)$$

5.3 Securing the Forward Handlers

There is another attack vector that the attacker can use during message dispatch: the user forwarder pointers (one for regular message dispatch and one for calls that return structures). The application can legitimately set these pointers using an API call, and many applications use this feature. We prevent attackers from modifying the two pointers by associating a HMAC with each pointer. The runtime updates the HMAC whenever it changes one of the pointers, and checks the HMAC before calling any of the handlers. We once again use a helper function:

$$H_2(fwd, K) = (fwd_L + K_L) * (fwd_H + K_H) \quad (4)$$

that combined with the table T gives us the HMAC:

$$H_{\text{fwd}}(fwd, K_{\text{fwd}}) = T[H_{2[31:(31-N)]}(fwd, K_{\text{fwd}})] \quad (5)$$

5.4 Secure Key Store

Our defense must keep several pieces of information secret to attackers: the HMAC keys—K_{class}, K_{cache}, and K_{fwd}—and the random table T. Discovering these values would allow the attacker to forge the HMAC values and bypass our defenses. It is therefore critical that we prevent attackers from disclosing or guessing these values.

To hide secrets from attackers, we rely on a security primitive known as XoM. This construct allows us to map virtual memory pages in memory so that they will generate a segmentation violation if accessed by anything other than the CPUs instruction fetch unit. Embedding secret values inside such pages allows the runtime to retrieve the values using function calls, while at the same time preventing attackers from reading the pages using direct information leaks. As outlined in Section 3, our threat model assume that one of the available XoM implementations [6, 9, 13, 21] has been deployed on the Objective-C runtime.

We store every secret value inside an execute-only accessor function that returns that secret value when called (the value itself is embedded in the body of the accessor). Additionally, the attacker cannot call the accessor, since that would require hijacking the control flow of the program.

Using one accessor per 64-bit secret value would increase memory usage significantly (we would need an 11-byte accessor for every 8-byte secret, producing a memory overhead of 37.5%), so we take another approach. We store the keys along with T inside a read-only memory region allocated at a random memory address (chosen randomly when calling `mmap`), then store a pointer to this region inside an accessor. To access the table, the runtime calls the accessor to get the pointer, then accesses T using a regular memory read.

To simplify our implementation and reduce the number of accessor calls, we store the HMAC keys as extra cells

(one per every 64 bits of key) inside the table T and perform a single accessor call to get the keys and table pointer. This lets the runtime retrieve all secret values using a single accessor call on the fast path, as opposed to one or more per key and then one for the table.

6 Evaluation

In the following, we discuss evaluation results related to the security and performance of our proposed defense.

6.1 Security

We evaluated the effectiveness of our defense using the proof-of-concept Subversive-C exploit described in Section 4. Our hardened runtime is a drop-in replacement which lets us keep all other parameters the same. Thus we can be confident that any differences during program execution are caused by our defense. When running our original attack without any adaptations, the program terminates either due to failing pointer checks (in most cases) or integrity checks. The reason for this is that our original attack does not generate all data structures touched by the integrity checks, but rather the bare minimum necessary to exploit message dispatch. Therefore some dereferenced pointers stay uninitialized. Even if an "accidental pointer" references valid memory, the actual integrity check fails due to the mismatch between computed and stored values.

Next, we extended our attack to generate all data structures that are needed for metadata verification, i.e., all structures that act as input for the HMAC. The easiest way to do so is to copy and then modify existing class structures. However, we were unable to compute the correct HMAC values used to secure the contents of both the cache and the method list. This left us with guessing the right value as the only remaining choice, which is difficult since we need to guess correctly for every counterfeit object in the chain. An incorrect guess for any object leads to detection and program termination.

In both cases Subversive-C is detected before any attacker-controlled code is executed. More specifically, program execution is aborted on the first message that is dispatched to a counterfeit object. As expected, we can create (valid) empty caches, or use the fallback mechanism of the cache protection which triggers a slow path lookup whenever the cache integrity check fails. Creating valid cache entries is difficult due to the keys used in the HMAC being inaccessible to the attacker. With the cache secured, we can try to forge the HMAC for the method list. Here we face even stronger security guarantees since we need to forge HMAC-MD5. Again the attacker lacks the knowledge of the input keys which are protected by the secure key store.

The third way to gain code execution would be to overwrite the forward handlers. However, even with an arbitrary write primitive to allow modifications of these handlers, this will not work. They are protected and the attacker again lacks the secret keys to generate valid handler entries.

We therefore conclude that our hardened runtime probabilistically detects and prevents Subversive-C exploits.

6.2 Performance

Since there is no standard set of Objective-C benchmarks, we compiled the following list of programs to evaluate the performance of our hardened runtime:

- **Dispatch** (micro) invokes a dynamically dispatched (and empty) method in a tight loop. The empty method is invoked 10^8 times.
- **Fibonacci** (micro) computes the 35th Fibonacci number using naive recursion.
- **Sorts**[4] (micro) uses different sorting algorithms (merge, quick, bubble, heap, insertion, selection, and the Objective-C library sort) to sort integer arrays of size 10^4. We combine the running times of all algorithms for our purpose.
- **XML parser**[5] (application) parses and creates song objects from XML data (100 or 1000 entries) using the `NSXMLParser` class [4] from the Objective-C standard library.
- **iTunes play** (application) plays a 5 second audio clip and closes iTunes.
- **iTunes encode** (application) converts a 4 minute song in MP3 format (7 MB) to M4A (7.6 MB) using the AAC encoder provided by iTunes.
- **Pages PDF** (application) exports a 100 page document (270 KB) to PDF (327 KB) in Pages (Apple's word processor).

When reporting results we average over 100 and 10 runs for micro benchmarks and application benchmarks, respectively. We automate the application benchmarks using AppleScript [3] which increases the consistency of our results and allows us to interact with real-world applications. Our hardened runtime is based on the Objective-C runtime version 532.2 (x86-64), which we use as the baseline for performance comparison. Experiments were conducted on an iMac 2.8 GHz Intel Core i7 with 8 GB memory running OS X Yosemite (10.10.5) and the latest versions of iTunes and Pages. In addition, we ran each benchmark with an instrumented version of our runtime to count the number of times the general dispatch function `msgSend` is invoked. Table 2 reports the results of our experiments. Note that the reported numbers do not include the overhead for the defenses assumed in Section 3.

The goal of the Dispatch benchmark is to give us an upper bound for the overhead incurred by our hardened

[4] The Sorts benchmark [24] was developed by Jesse Squires.

[5] XML parser is an adaptation of a benchmark from Apple [5] that compares the performance of XML parsing libraries on iOS.

Benchmark	`msgSend` calls	Calls/ms	Overhead
Dispatch	10,000,000,215	190583	106.46 %
Fibonacci	2,986,070,515	173527	88.66 %
Sorts	13,329,480,611	82597	34.54 %
Average (micro)		148902	**76.55 %**
XML-100	7,940,898	6475	2.81 %
XML-1000	78,119,698	6386	1.97 %
iTunes play	8,592,257	1667	0.37 %
iTunes enc.	114,948	29	1.82 %
Pages PDF	78,691	46	0.75 %
Average (application)		2921	**1.54 %**

Table 2: `msgSend` invocation counts and overheads.

runtime. This is realistic since the benchmark does no real work and just calls an empty method repeatedly. This puts maximum pressure onto the message dispatch mechanism which is the only part of the runtime affected by our protection scheme. Using the data from Table 2 we conclude that the maximum slowdown is bounded by 2x.

The Fibonacci benchmark mainly executes recursive method calls plus an add operation and some control flow to terminate recursion. Note that we mean dynamically dispatched call, i.e., calls dispatched via `msgSend`, whenever we write method call in this section. Standard C function calls are valid in Objective-C, but do not go through `msgSend`. Therefore our defense does not reduce the performance of regular calls to C functions.

The Sorts benchmark is implemented in a way that leads to a high number of `msgSend` calls. Rather than using plain integer arrays, it uses Objective-C collections that require boxing of the integer numbers they store. So what normally is a simple array access becomes two method calls: one to index the collection and one to unbox the integer for comparison. The benchmark results reflect this accordingly. To back our claim we modified the benchmark to use plain integer arrays, replacing `NSMutableArray` with (`int arr[], int len`). As expected, the difference in running times then falls into the range of measurement noise ($< 1\%$).

At this point we want to draw attention to the relation between `msgSend` calls per millisecond and the reported overhead. For compute-intensive programs it is directly proportional. In other words: the more real work a program does, the smaller the overhead.

With the second set of benchmarks we want to demonstrate that although overhead for individual micro benchmarks is considerable, it is insignificant in practice. Especially for interactive applications like iTunes and Pages there is no perceivable slowdown during normal use. For the benchmarks iTunes play and Pages PDF the reported overhead is in the range of measurement noise. Our explanation is that Objective-C is mostly used to implement an application's logic and user interface while core functionality (playing and encoding music files, exporting to PDF) is provided by C libraries. Hence, we incur little to no overhead on those activities. The only time an end user experiences additional delay is during program startup. Table 3 quantifies this delay.

Benchmark		HelloWorld	iTunes
Startup	Base	35	1020
	Hardened	107	1478
Overhead	**Total**	**72**	**458**
	Random table	43	43
	Integrity checks	29	415

Table 3: Startup times and overhead in milliseconds.

We measured the running time of a simple HelloWorld program and the startup time of iTunes both with our baseline and hardened runtime. The total startup overhead for HelloWorld is 72 ms, whereof 43 ms are attributed to seeding the random table which aids the implementation of the secure key store. The remaining 29 ms are spent to populate and check hashes of 280 core classes, e.g., `NSObject`, which are eagerly initialized by the runtime. The time needed to seed the random table depends linearly on the size of the table. In our implementation the table holds 1 million keys resulting in 8 MB total size. The size of the table can be adjusted to adhere to an application's security and memory constraints.

For iTunes the total startup overhead is about half a second. This is due to iTunes being a complex application initializing roughly 2000 Objective-C classes during startup. Considering typical application usage patterns we argue that this is acceptable since there is no further perceived slowdown during continued use.

7 Discussion

In Section 5.4, we presented our approach to securing the key store against leaks: we store its contents at a random address in memory, then store the address as a pointer inside XoM. Since the pointer is stored in a single non-readable location in memory, attackers cannot use an information leak attack to locate the table. However, this approach could expose the table to attackers in some other way, e.g., probing all memory pages one by one to find the table. However, probing attacks would be difficult for two reasons. First, locating all readable virtual memory pages is difficult, assuming attackers cannot install a signal handler or obtain a virtual memory map for the program. Second, to identify the table T in memory, attackers would need to distinguish between randomly-generated bytes and proper program data. Therefore, the barriers to attackers locating T are high. Choosing whether to store the random table in execute-only or readable memory

presents a potential security vs. memory usage trade-off. Storing T directly in XoM provides guaranteed secrecy, at the cost of an extra 37.5% memory usage for the table. We therefore leave this decision to system developers.

Side-channel attacks are another potential class of attacks against the key store, or more specifically against the table T. For example, attackers could derive the indices used to access the table, and therefore the values of H_1, by measuring the externally-visible metrics (such as cache misses or CPU cycles) while the runtime performs its integrity checks (similar attacks have been demonstrated on cryptographic functions [41]). If such attacks prove feasible and likely, the same defenses that protect cryptographic algorithms can also be applied to our key store [12].

One other interesting mitigation is object layout randomization. In the runtime, the offsets of instance variables from the start of an object are dynamically defined when its class is loaded. The Objective-C language puts no constraints on the order of variables inside an object, i.e., there is no requirement that they be in the same order that they appear in the source code. Therefore, it would be possible to randomize the object layout. This would not defend against an attacker who can read all of memory, but would make it harder to inject counterfeit objects.

8 Related Work

The work on exploitation and exploit mitigations is extensive. Due to the page limitation, we refer the interested reader to Szekeres et al. [39] and focus on recent, closely related work on attacks and defenses.

Attacks Our work is inspired by the recently published COOP technique [14, 31]. COOP itself is but one of a series of exploitation techniques that are able to bypass coarse-grained CFI policies [10, 16, 23, 32]. By virtue of exploiting the dynamic dispatching mechanisms, both Subversive-C and COOP-style attacks are not stopped by randomization-based defenses that have been widely studied in the last years [26]. RILC is another related exploitation technique [28, 40]. Whereas Subversive-C and COOP reuse dynamically bound methods, RILC reuses dynamically linked functions in the procedure linkage table such as those in the C standard library. Despite the name, RILC applies to other libraries than `libc` [37].

Defenses MoCFI [15] was designed to protect Objective-C code running on iOS for ARM.[6] MoCFI maintains a shadow stack to enforce that a return targets its original caller. Further, forward indirect branches must follow a valid CFG path calculated by means of static analysis. However, Subversive-C circumvents these protections: (1) it never violates call-return matching, and (2) it dispatches all malicious function calls via `msgSend` which resembles a valid CFG path. Further, MoCFI's protection of the `msgSend` selector and class metadata do not prevent Subversive-C since we do not corrupt selectors and avoid changing class structures in ways that MoCFI would detect. Specifically, MoCFI ensures that the class or superclass pointer for each object is known and prevents creation of entirely new classes at run time. However, MoCFI must allow new class structs, where only the superclass pointer is known to MoCFI. As a result, we can construct Subversive-C attacks that use valid superclass pointers or alter the method lists of existing classes. We stress that MoCFI and our novel defense complement each other and can prevent a broader range of attacks when used in concert.

CFR [29] is a compiler-based CFI implementation for Objective-C code on iOS. Unlike MoCFI, which protects returns using a shadow stack, Control-Flow Restrictor (CFR) enforces a purely static policy for all indirect branches. Since CFR does not place any particular restriction on calls dispatched via `msgSend`, CFR does not stop Subversive-C attacks but could complement our defense just like MoCFI. CFR does support programmer-inserted annotations to further constrain the CFG which could potentially prevent our attack; doing so requires manual effort and may lead to errors that prevent programs from running correctly.

Readactor++ [14] is the first randomization-based defense which thwarts COOP attacks by randomizing and booby trapping C++ vtables. Due to the differences in dispatching mechanisms, the concepts behind Readactor++ does not generalize to prevent Subversive-C exploits. For example, vtables are immutable and can be hidden using XoM whereas Objective-C class metadata is mutable which is why we opted to use HMACs instead.

CPI [25] separates regular data from control data which thwarts Subversive-C exploits. CPI relies on static analysis to identify sensitive data which is more challenging for Objective-C code than C and C++ code. It also requires software-fault isolation or hardware segmentation to resist memory probing attacks [19].

Recently, van der Veen et al. [42] presented a purely binary-based defense against COOP, which breaks data flows between vfgadgets through argument registers. Their method enforces a CFI policy derived via static code analysis that limits the vfgadget space available to an attacker, thus making attacks harder. As our example exploit relies on data flows through argument registers, it would be thwarted by this defense. However, we note that Subversive-C—and also COOP in general—does not inherently require register-based data flows, as attackers can potentially fall back to leveraging overlapping counterfeit objects only or passing data via scratch areas.

[6] Our research uses but is not specific to x86-64 hardware.

Similar to our defense, CryptoCFI [27] uses HMACs to protect sensitive pointers. CryptoCFI computes cryptographically secure HMACs using special AES instructions on the latest Intel x86 processors. Although a direct comparison is not possible, the overheads of using this defense is likely far higher than ours and requires that part of the SIMD register file be reserved for CryptoCFI.

9 Conclusion

This paper presented Subversive-C which is the first whole-function reuse attacks abusing the `msgSend` feature of Objective-C. Our attack shows that COOP-style attacks which are far harder to prevent than ROP-style code-reuse, are not limited to C++ code. We discuss the intricacies of Objective-C message dispatch and how to utilize them for our attack. Specifically, we describe an attack targeting the AppKit (x86-64) library for OS X, which is a core building block for many popular applications. Finally, we present a practical defense against Subversive-C and show that it imposes a negligible performance overhead when protecting real-world applications.

Acknowledgments

We would like to thank David Chisnall, our shepherd Randal Burns, and the anonymous reviewers for their valuable input. Fabian Kammel helped with the implementation of the attack outlined in Section 4.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237, by the National Science Foundation under award number CNS-1513837, by the ERC Starting Grant No. 640110 (BASTION) as well as gifts from Mozilla, Oracle and Qualcomm. In addition, this work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, Ú., AND LIGATTI, J. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security 13* (2009).

[2] ALEPH ONE. Smashing the stack for fun and profit. *Phrack Magazine 7* (1996).

[3] APPLE INC. AppleScript language guide. `https://developer.apple.com/library/mac/documentation/AppleScript/Conceptual/AppleScriptLangGuide/introduction/ASLR_intro.html`, 2015.

[4] APPLE INC. NSXMLParser class reference. `https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSXMLParser_Class`, 2015.

[5] APPLE INC. XMLPerformance on iOS. `https://developer.apple.com/library/ios/samplecode/XMLPerformance/Introduction/Intro.html`, 2015.

[6] BACKES, M., HOLZ, T., KOLLENDA, B., KOPPE, P., NÜRNBERGER, S., AND PEWNY, J. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[7] BELLARE, M. New proofs for NMAC and HMAC: Security without collision-resistance. In *Advances in Cryptology - CRYPTO 2006*, C. Dwork, Ed., vol. 4117 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 602–619.

[8] BLACK, J., HALEVI, S., KRAWCZYK, H., KROVETZ, T., AND ROGAWAY, P. UMAC: Fast and secure message authentication. In *Advances in Cryptology—CRYPTO* (1999).

[9] BRADEN, K., CRANE, S., DAVI, L., FRANZ, M., LARSEN, P., LIEBCHEN, C., AND SADEGHI, A.-R. Leakage-resilient layout randomization for mobile devices. In *Symposium on Network and Distributed System Security (NDSS)* (2016), NDSS.

[10] CARLINI, N., AND WAGNER, D. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium* (2014).

[11] CHECKOWAY, S., DAVI, L., DMITRIENKO, A., SADEGHI, A., SHACHAM, H., AND WINANDY, M. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[12] CRANE, S., HOMESCU, A., BRUNTHALER, S., LARSEN, P., AND FRANZ, M. Thwarting cache side-channel attacks through dynamic software diversity. In *Symposium on Network and Distributed System Security (NDSS)* (2015).

[13] CRANE, S., LIEBCHEN, C., HOMESCU, A., DAVI, L., LARSEN, P., SADEGHI, A.-R., BRUNTHALER, S., AND FRANZ, M. Readactor: Practical code randomization resilient to memory disclosure. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[14] CRANE, S., VOLCKAERT, S., SCHUSTER, F., LIEBCHEN, C., LARSEN, P., DAVI, L., SADEGHI, A.-R., HOLZ, T., SUTTER, B. D., AND FRANZ, M. It's a TRAP: Table randomization and protection against function reuse attacks. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[15] DAVI, L., DMITRIENKO, A., EGELE, M., FISCHER, T., HOLZ, T., HUND, R., NÜRNBERGER, S., AND SADEGHI, A.-R. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *NDSS* (2012).

[16] DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium* (2014).

[17] DAVI, L., SADEGHI, A.-R., AND WINANDY, M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011).

[18] DE MOURA, L., AND BJØRNER, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[19] EVANS, I., FINGERET, S., GONZALEZ, J., OTGONBAATAR, U., TANG, T., SHROBE, H., SIDIROGLOU-DOUSKOS, S., RINARD, M., AND OKHRAVI, H. Missing the point: On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[20] FRANTZEN, M., AND SHUEY, M. StackGhost: Hardware facilitated stack protection. In *USENIX Security Symposium* (2001).

[21] GIONTA, J., ENCK, W., AND NING, P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *ACM Conference on Data and Application Security and Privacy (CODASPY)* (2015).

[22] GÖKTAS, E., ATHANASOPOULOS, E., BOS, H., AND PORTOKALIDIS, G. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[23] GÖKTAS, E., ATHANASOPOULOS, E., POLYCHRONAKIS, M., BOS, H., AND PORTOKALIDIS, G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security Symposium* (2014).

[24] JESSE SQUIRES. Objective-c sorts. `https://github.com/jessesquires/objc-sorts`, 2014.

[25] KUZNETSOV, V., SZEKERES, L., PAYER, M., CANDEA, G., SEKAR, R., AND SONG, D. Code-pointer integrity. In *USENIX Security Symposium* (2014).

[26] LARSEN, P., HOMESCU, A., BRUNTHALER, S., AND FRANZ, M. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)* (2014).

[27] MASHTIZADEH, A. J., BITTAU, A., BONEH, D., AND MAZIÈRES, D. CCFI: Cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)* (2015).

[28] NERGAL. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine 11* (2001).

[29] PEWNY, J., AND HOLZ, T. Control-flow Restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)* (2013).

[30] ROEMER, R., BUCHANAN, E., SHACHAM, H., AND SAVAGE, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information System Security 15* (2012).

[31] SCHUSTER, F., TENDYCK, T., LIEBCHEN, C., DAVI, L., SADEGHI, A.-R., AND HOLZ, T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE Symposium on Security and Privacy (S&P)* (2015).

[32] SCHUSTER, F., TENDYCK, T., PEWNY, J., MAASS, A., STEEGMANNS, M., CONTAG, M., AND HOLZ, T. Evaluating the effectiveness of current anti-ROP defenses. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2014).

[33] SERNA, F. J. The info leak era on software exploitation. In *BlackHat USA* (2012).

[34] SHACHAM, H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)* (2007).

[35] SHACHAM, H., PAGE, M., PFAFF, B., GOH, E., MODADUGU, N., AND BONEH, D. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)* (2004).

[36] SIEBERT, J., OKHRAVI, H., AND SÖDERSTRÖM, E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM Conference on Computer and Communications Security (CCS)* (2014).

[37] SKOWYRA, R., CASTEEL, K., OKHRAVI, H., AND ZELDOVICH, N. Systematic analysis of defenses against return-oriented programming. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2013).

[38] SNOW, K. Z., MONROSE, F., DAVI, L., DMITRIENKO, A., LIEBCHEN, C., AND SADEGHI, A. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[39] SZEKERES, L., PAYER, M., WEI, T., AND SONG, D. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[40] TRAN, M., ETHERIDGE, M., BLETSCH, T., JIANG, X., FREEH, V. W., AND NING, P. On the expressiveness of return-into-libc attacks. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)* (2011).

[41] TROMER, E., OSVIK, D. A., AND SHAMIR, A. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology* (2010).

[42] VAN DER VEEN, V., GÖKTAS, E., CONTAG, M., PAWLOWSKI, A., CHEN, X., RAWAT, S., BOS, H., HOLZ, T., ATHANASOPOULOS, E., AND GIUFFRIDA, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE Symposium on Security and Privacy (S&P)* (2016).

[43] ZHANG, C., WEI, T., CHEN, Z., DUAN, L., SZEKERES, L., MCCAMANT, S., SONG, D., AND ZOU, W. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)* (2013).

[44] ZHANG, M., AND SEKAR, R. Control flow integrity for COTS binaries. In *USENIX Security Symposium* (2013).

Callinicos: Robust Transactional Storage for Distributed Data Structures

Ricardo Padilha Enrique Fynn Robert Soulé Fernando Pedone
Universitá della Svizzera italiana (USI)
Switzerland

Abstract

This paper presents Callinicos, a robust storage system with a novel transaction protocol that generalizes mini-transactions. This protocol allows Callinicos to cope with Byzantine failures, support cross-partition communication with transactions, and implement on-demand contention management. We have evaluated Callinicos with a set of micro-benchmarks, and two realistic applications: a Twitter-like social network and a distributed message queue. Our experiments show that: (i) cross-partition communication improves performance by reducing the number of aborts, and (ii) the conflict resolution protocol results in no aborts in the presence of contention and no overhead in the absence of contention.

1 Introduction

Many application domains including retail, healthcare, and finance, have a need for storage systems that tolerate failures and scale performance without sacrificing consistency. However, designing systems that satisfy these requirements is a daunting task. Among the various approaches proposed in recent years, mini-transactions strike an attractive balance between functionality and performance [2]. Mini-transactions allow applications to atomically access and conditionally modify distributed data. They are also amenable to optimizations that result in scalable performance, such as update batching, message piggybacking, and state partitioning.

Like database transactions, mini-transactions hide the complexities that result from concurrent execution and failures. However, mini-transactions optimize the execution and commit of transactions by exposing a restricted set of operations. Thes operations allow a transaction to read a storage entry; compare an entry with a value (i.e., equality comparison); and update the value of an entry. A mini-transaction is only committed if all its compare operations are successful. As a consequence of these restrictions, mini-transactions can piggyback the last transaction action onto the first phase of the two-phase commit, saving a network round-trip. Despite their simple execution model, several non-trivial applications have been developed with mini-transactions, including a cluster file system and a group communication service.

Yet, despite the fact that several storage systems from the research community have proposed the use of mini-transactions [2, 34, 35], few real-world applications have employed them. This paper argues that mini-transactions suffer from several problems that have hindered their wide-spread adoption: (i) they provide only limited reliability, (ii) they disallow indirect memory accesses and prevent data from transferring from one partition to another within a transaction, and (iii) for workloads with high-contention (e.g., hot spots), they are affected by frequent aborts which leads to increased latencies. Below, we discuss each of these issues in more detail.

Limited reliability. Although mini-transactions can tolerate benign failures, such as crash failures, there are many other types of failures that occur in data centers. Byzantine failures are a common occurrence [16], resulting from any number of reasons including disk faults [8, 38], file system bugs [50], or human errors [23]. Mission-critical services require strong end-to-end guarantees that hold all the way up to the application layer.

Restricted operations. To provide scalable performance, mini-transactions disallow indirect memory accesses and prevent data from transferring from one partition to another within a single transaction. This restriction significantly impacts how applications can implement many common, but surprisingly simple operations. As a motivating example, consider the pseudocode to atomically swap the contents of two stored keys shown in Figure 1 (left). The operations could not be implemented within a single mini-transaction, since key_a and key_b may be stored in different partitions and there is no way for partitions to exchange information within a transaction. The only way one could implement the above code would

```
swap_tx(key_a, key_b) {
    x ← read(key_a)
    y ← read(key_b)
    write(key_a, y)
    write(key_b, x)
}
```

```
swap_tx1(key_a, key_b) {
    x ← read(key_a)
    y ← read(key_b)
} // return x, y
swap_tx2(key_a, key_b, x, y) {
    cmp(key_a, x)
    cmp(key_b, y)
    write(key_a, x)
    write(key_b, y)
}
```

Figure 1: A transaction to swap two entries (left) and its schematic implementation using mini-transactions (right).

		Contention management	
		Without	With
Failure Model	Benign	Sinfonia [2]	Rococo [34] H-Store [43] Calvin [44]
	Byzantine	Augustus [35]	**Callinicos**

Table 1: Overview of transactional key-value stores.

be to split the *swap* operation in two independent transactions, *swap_tx1* that reads both keys and *swap_tx2* that updates them. However, with two transactions, the *swap* operation would no longer be atomic and a third transaction could execute between *swap_tx1* and *swap_tx2*. This third transaction could change either of the values, which would lead to a non-serializable execution. One could extend *swap_tx2* with the mini-transaction compare operation (*cmp*) to conditionally write the new values if they have not been modified, and abort otherwise, as shown in Figure 1 (right). Under high-contention workloads, such an implementation would lead to frequent aborts, decreasing storage throughput and increasing latency.

Contention. Mini-transactions rely on optimistic concurrency control (e.g., [12, 25, 27, 40, 41]). Unfortunately, in the presence of contention, optimistic concurrency control leads to aborts, poor use of system resources and increased latencies. It also put additional burden on developers, since key-value stores with optimistic concurrency control often delegate the task of managing contention to application designers.

This paper presents Callinicos, a storage system designed to address the limitations of mini-transactions. While prior work has addressed some of the above challenges individually, no system at present addresses all of them (see Table 1). Augustus [35] augments mini-transactions with support for Byzantine fault tolerance. Just like Sinfonia's mini-transactions [2], Augustus's transactions are prone to aborts in the presence of contention. Calvin [44], Rococo [34], and H-Store [43] account for contention at different degrees, as we detail in § 6, but none of these systems can cope with Byzantine failures. In contrast, Callinicos tolerates the most extreme types of failure, is not constrained by restricted operations, and never aborts transactions due to contention.

Callinicos implements *armored-transactions*, a novel transaction model that can split the execution of an armored-transaction in rounds to cope with data contention and allow partitions to exchange data. Armored-transactions execute in a single round, like mini-transactions, if they are not subject to data contention and cross-partition communication. Additional rounds are used to order armored-transactions that conflict and exchange data across partitions within a single and atomic armored-transaction. Armored-transactions can be viewed as a generalization of basic mini-transactions, which allow for cross-partition communication, and implement on-demand contention management.

Using Callinicos, we have implemented two real-world applications: *Buzzer*, a twitter clone, and *Kassia* a distributed message queue based on Apache Kafka. Although it would have been possible to implement these applications using mini-transactions, their design would be significantly more complicated and the performance would be poor, as we show in § 5. We have evaluated Callinicos under a variety of deployment and workload scenarios, and our experiments show that in the absence of contention, Callinicos introduces no overhead in the execution of transactions and can scale throughput with the number of nodes; in the presence of contention it orders transactions to avoid aborts. Overall, this paper makes the following contributions:

- It details a novel multi-round transactional execution model that generalizes the mini-transaction model, while allowing for cross-partition data exchange and contention management.
- It describes the design and implementation of a robust, distributed storage system built using the new transactional execution model.
- It demonstrates through experimental evaluation that the transactional model and storage system offer significant performance improvements over mini-transactions.

The remainder of the paper is structured as follows: it details the system model (§ 2), discusses Callinicos's design (§ 3), and presents the transaction execution protocol (§ 4). Then it evaluates the performance (§ 5), reviews related work (§ 6), and concludes (§ 7).

2 System model and definitions

We consider a distributed system with processes that communicate by message passing. Processes do not have access to a shared memory or a global clock. The system is asynchronous (i.e., no bounds on processing times or message delays) and there is an arbitrary number of clients and a fixed number n of servers, where clients and servers are disjoint.

Processes can be *correct* or *faulty*. A correct process follows its specification; a faulty, or Byzantine, process presents arbitrary behavior. There is a bounded although arbitrary number of faulty clients. Servers are divided into disjoint groups. Each group g contains n_g servers, out of which f_g can be faulty.

Processes communicate using either one-to-one or one-to-many communication. One-to-one communication guarantees that if sender and receiver are correct, then every message sent is eventually received. One-to-many communication is based on atomic multicast and ensures that: (a) a message multicast by a correct process to group g will be delivered by all correct processes in g; (b) if a correct process in g delivers m, then all correct processes in g deliver m; and (c) every two correct processes in g deliver messages in the same order.

Atomic multicast algorithms need additional assumptions in the system model (e.g., partial synchrony [21]). These assumptions are not explicitly used by our protocols. While several BFT protocols implement the atomic multicast properties enumerated above (e.g., [10, 26]), we assume (and have implemented) PBFT [14], which can deliver messages in four communication steps and requires $n_g = 3f_g + 1$ servers.

We use SHA-1 based HMACs for authentication, and AES-128 for transport encryption. We assume that adversaries (and faulty processes under their control) are computationally bound and unable, with very high probability, to subvert the cryptographic techniques used. Adversaries can coordinate faulty processes and delay correct processes in order to cause the most damage to the system. Adversaries cannot, however, delay correct processes indefinitely.

3 Storage Design

Callinicos is a distributed key-value store with support for transactions. Like other systems that implement mini-transactions, Callinicos uses state partitioning to improve scalability. While the partitioning algorithm has an impact on performance, the choice of partitioning scheme is orthogonal to the design of Callinicos.

Clients have access to a *partition oracle*, which knows the partitioning scheme used for a particular deployment. The partition oracle can be implemented using a variety of designs: centralized, replicated, fully-distributed, etc. Each design has different tradeoffs for performance overhead on the system. Our Callinicos prototype uses static partitioning and each client has a-priori knowledge of the partitioning scheme.

$l \in Lit$	*Literals*
$k \in \mathcal{K}$	*Keys*
$v \in ID$	*Identifiers*
$t ::= s$	Transaction
$s ::= s_1; s_2$	Sequence
$\mid v = e$	Assignment
$\mid$ if e then s_1 else s_2	Conditional Branch
$\mid$ while e do s	While Loop
$\mid r \mid w \mid c$	Transaction Operators
$e ::= e_1 \&\& e_2 \mid e_1 \mid\mid e_2 \mid !\, e_1$	Logical Expr.
$\mid e_1 > e_2 \mid e_1 < e_2 \mid e_1 == e_2$	Relational Expr.
$\mid e_1 >= e_2 \mid e_1 <= e_2 \mid e_1 != e_2$	
$\mid e_1 * e_2 \mid e_1 / e_2$	Multiplicative Expr.
$\mid e_1 + e_2 \mid e_1 - e_2$	Additive Expr.
$\mid v \mid l$	Variable or Literal
$r ::= read(k)$	Read
$w ::= write(k, v) \mid delete(k)$	Update
$c ::= export(id) \mid import(id) \mid rollback$	Control

Figure 2: Transaction language syntax (subset).

3.1 Unrestricted operations

Clients access the storage by means of pre-declared transactions, similar to stored procedures in relational databases. These transactions, named armored-transactions, are written in a small transaction language, designed to meet three goals. First, it is a subset of popular general purpose programming languages, such as Java or C. This means that the syntax is familiar to developers, and the transaction specification could be easily embedded in larger programs. Second, the subset is large enough to support the basic operations required for expressive, multi-partition transactions. Third, the language makes explicit the points in the code where data may cross partition boundaries.

The language, whose syntax is in Figure 2, includes support for variable declarations and assignment; logical, relational, and arithmetic expressions; and basic control flow for branching and looping. Statements and expressions may be composed to express more complex logic.

At the heart of the transaction language are a set of built-in operations that are used to manipulate storage entries. These operations include *read* and *write* operations, as well as *delete*, to remove a key from the store.

Transaction-local variables persist for the duration of the transaction, and can be shared between partitions by using *export* and *import* control operations, which we explain in the next section. A transaction can request to

abort its execution with a *rollback* operation.

Callinicos guarantees strict serializability for update transactions from all clients and read-only transactions submitted by correct clients. An update transaction contains at least one operation that modifies the state. Callinicos provides no guarantees for read-only transactions from faulty clients. More precisely, for every history H representing an execution containing committed update transactions and committed read-only transactions submitted by correct clients, there is a serial history H_s containing the same transactions such that (a) if transaction T reads an entry from transaction T' in H, T reads the same entry from T' in H_s; and (b) if T terminates before T' starts in H, then T precedes T' in H_s.

3.2 Byzantine fault tolerance

Both clients and servers can present faulty behavior. To cope with faulty servers, each partition is fully replicated on a group of servers. Each server group uses atomic multicast and state machine replication [28, 39], implemented with PBFT [14], to ensure consistency. Therefore, although partitions can contain faulty servers, the partition as a whole follows the Callinicos protocol.

Faulty clients can attempt to disrupt the execution with a number of attacks, which include (a) leaving a transaction unfinished in one or more partitions, (b) forcing early termination of honest transactions (i.e., from a correct client), and (c) attempting denial-of-service attacks. We describe these attacks and the mechanisms to counter them in more detail in § 4.3.

3.3 Contention management

Callinicos implements on demand contention management. Every armored-transaction is multicast to each partition involved in the armored-transaction. The involved partitions are computed from the pre-declared armored-transaction, possibly reaching all partitions if the appropriate subset cannot be statically determined. Servers in a partition then try to acquire all the locks required by the armored-transaction. If all locks cannot be acquired, the servers request the ordering of the armored-transaction. This mechanism strives to avoid the overhead of ordering in the absence of contention, typical of distributed locking, and the penalty of aborts, typical of optimistic concurrency control.

4 Armored Transactions

Armored transactions implement a novel *multi-round* protocol, which generalizes the concept of mini-transactions by allowing data to flow across partitions in the context of the same transaction. The exchange of data between partitions in armored transactions is mediated by the clients that submit these transactions. Servers from each partition send the data to be transferred to the client at the end of a round, and the client forwards the data to the proper destination at the beginning of the next round. Figure 3 shows the complete execution for armored transactions. We explain the details in the following sections.

4.1 Transaction pre-processing

Before a client application can submit an armored-transaction for execution on the server, it first must transform the armored-transaction expressed in the Callinicos transaction language into a *transaction matrix*. A transaction matrix is a representation of the armored-transaction in which operations have been assigned to specific *partitions* and specific *rounds*. Each column in a transaction matrix represents a partition-specific sequence of rounds. Each row in the matrix represents the operations that will be executed in a single round. Intuitively, operations are mapped to a partition if they read or write data from that partition. Two operations, o_1 and o_2 are mapped to separate rounds if o_2 depends on the output of o_1, and they are executed on separate partitions.

Assuming a two-partition deployment, where key_a and key_b are on partitions 1 and 2, respectively, the transaction matrix for the swap operation shown in § 1 will be a 2×2 matrix (see Table 2). Both reads in the swap matrix are independent of any other operations and executed in the first round. Since the writes depend on the result of the reads, they are placed on the second round.

	Partition 1	Partition 2
Round 1	$a \leftarrow read(key_a)$ $export(a)$	$b \leftarrow read(key_b)$ $export(b)$
Round 2	$import(b)$ $write(key_a, b)$	$import(a)$ $write(key_b, a)$

Table 2: Transaction matrix for *Swap* command.

To build a transaction matrix, clients need to determine two pieces of information: (i) the data dependencies between operations, and (ii) the partitions where data is stored. Information about data dependencies can be learned by performing a static analysis of the armored-transaction logic to track data flow. Information about partition mappings can be learned by consulting the partition oracle. If the value of a key is known statically, then the partition oracle will tell the client where a key is stored. If the value of a key can only be determined at runtime, then the oracle will tell the client that no assertion about the data layout can be made, and instruct the

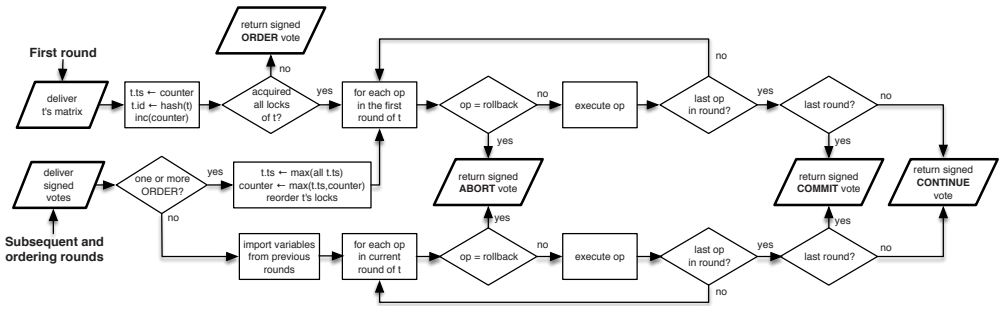


Figure 3: Armored-transaction execution on the servers in Callinicos.

client to execute a guarded version of the operation on all partitions (e.g., *if contains(k) then op(k)*).

Clients organize the operations in columns and rows as follows: (1) Operations that are confined to specific partitions are placed on their respective columns, or are executed on all partitions. (2) Operations that depend on the outcome of a previous operation executed on a different partition are placed on the row immediately after the row of the operation they depend on. (3) Independent operations are executed as early as possible.

After operators are assigned to rows and columns, each entry is augmented with *export* and *import* operations, which transport data across partitions at the end and beginning of each round, respectively. The details of the *export* and *import* operations, and the data transfer mechanism between rounds are presented in detail in § 4.2.

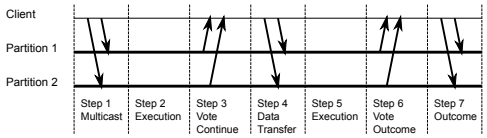
To reduce the implementation effort, our prototype relies on hand-written annotations that indicate partition and round number for each line of code in the transaction language specification. In a continuing development effort, we are working to automate the transformation.

4.2 Execution under normal conditions

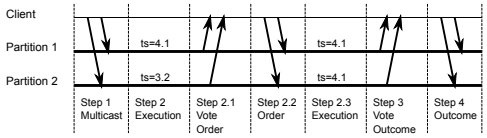
Once delivered for execution, an armored-transaction first transitions to a transient state (i.e., *ordering* or *pending*) and eventually reaches a final state (i.e., *committed* or *failed*). All state transitions are irreversible, irrevocable, and can be proved by signed certificates, which are generated as part of the state transition.

State transitions happen as the result of the *round executions*. Each round is composed of a request-reply exchange between a client and the servers. Requests from clients can start a new transaction, continue an ongoing transaction, or finalize a transaction. Replies from servers contain a signed vote with the round number, the outcome of the round, the current *timestamp* of the armored-transaction, and any applicable execution results. Servers implement a local and unique *counter*, used to assign timestamps to transactions, as described next. In the absence of failures and malicious behavior, an armored-transaction follows the sequence of steps described next. We discuss execution under failures in § 4.3.

Transaction submission. A client submits a new armored-transaction t by multicasting t's transaction matrix to each partition g involved in t. One multicast call is done per partition (see Figure 4, Step 1). Since the matrix is fully defined before it is multicast, every server knows how many rounds are needed to complete the transaction, which data should be returned in each round, and which data will be received in each round.



(a) with data transfer.



(b) with ordering

Figure 4: Execution of a two-round transaction.

The first round. Once server s delivers t's matrix, t becomes *delivered* at s and the value of s's counter becomes the timestamp ts of t. Each correct server s tries to acquire all the locks required by t (Figure 4, Step 2). If the lock acquisition fails (i.e., another transaction has a conflicting lock), s stops the execution of t and issues a signed vote for ORDER (Figure 4b, Step 2.1), meaning that s requires the next round of t to be an *ordering round* (i.e., t is in the *ordering* state at s). If s can acquire all locks needed by t, s will execute t's operations that belong to the current round. If s finds a rollback operation, it stops the execution of t, issues a signed vote for ABORT, meaning that it expects the current round to be the last and the next multicast from the client to provide a *termination certifi-*

cate. If *s* completes *t*'s execution in the current round and the round is the last, *s* issues a signed vote for COMMIT (Figure 4a, Step 6). If the current round is not the last, *s* issues a signed vote for CONTINUE (Figure 4a, Step 3). In the case of a vote for CONTINUE, if the transaction contains any *export* operations in the current round, the respective variables are included in the signed vote. For any of these three votes *t* also becomes *pending*. In any case, a server's vote on the outcome of a round is final and cannot be changed once cast.

Lock management. A transaction can request *retrieve* and *update* locks on single keys and ranges of keys. Both single-key and key-range locks keep track of the acquisition order in a *locking queue*, i.e., a queue containing all the transactions that requested that specific lock, ordered by increasing timestamp. Retrieve locks can be shared; update locks are exclusive. A requested lock that conflicts with locks already in place will trigger an ordering round. Lock acquisition is performed sequentially, by a single thread, and at once for the complete transaction matrix, i.e., each server traverses its entire column and tries to acquire locks on all keys defined in all rounds. If a round performs retrieve or update operations on a key that is the result of a previous round, the server must perform a partition-wide lock. Callinicos's execution model avoids deadlocks since all the locks of a transaction are acquired atomically, possibly after ordering (defined next). If two conflicting multi-partition transactions are delivered in different orders in two different partitions, then each transaction will receive at least one vote to ORDER from one of the involved partitions, which will order both transactions.

Subsequent rounds. The execution of each following round starts with the client multicasting the vote certificate from the previous round (Figure 4a, Step 4). Each server then validates the signatures, tallies the votes, and proceeds according to the outcome of the certificate. The possible outcomes of the certificate are either: COMMIT, ABORT, ORDER, or CONTINUE.

If at least one vote is for COMMIT, or ABORT, then the transaction will terminate, as described below. If no partition votes for ORDER, *t* will transition to the *pending* state. Otherwise, *t* will go through an *ordering round* before transitioning to *pending* (Figure 4b, Step 2.2). The *pending* state indicates that a transaction has not yet been finalized (i.e., it has neither been committed nor aborted).

A vote for CONTINUE indicates that the previous round was not the last round of *t*. The transaction will remain in the *pending* state, and *s* executes the next round of *t*. The execution of the next round starts with the importing of variables exported in the previous round as indicated by *export* operations. These values are stored locally for the duration of the transaction, and are used in subsequent rounds. Once the execution of the round completes, the server issues a signed vote following the same rules as the first round. A server will not execute subsequent rounds until it has been presented with a vote certificate for the round it just executed. Furthermore, a server will not re-execute a round for which it has already issued a vote, unless it is an *ordering round*, as we explain next.

The ordering round. To order a transaction, the server examines the timestamps of all votes, selects the highest value, *maxts*, and sets *t*'s timestamp to *maxts*. If *s*'s counter is lower than *maxts*, *s* sets its counter to *maxts*. Then, *t*'s locks are re-ordered and *t* is scheduled for re-execution (Figure 4b, Step 2.3). Because all (correct) servers in the partitions that contain *t* participate in the same protocol steps, *t* will have the same timestamp in all involved partitions. The re-ordering of *t*'s locks ensures that for all locks ℓ acquired by *t*, there is no armored-transaction *u* with a timestamp lower than *t* ordered after *t* in ℓ's locking queue. We explain the re-ordering of locks below. Since a change of execution order may lead to a different outcome, *t* is re-executed when it becomes the transaction with the lowest timestamp that has no conflicts. The ordering round, if present, will always be the second round since lock acquisition is done in the first round. As a corollary, correct servers only vote for ORDER as the outcome of the first round. Furthermore, after an ordering round *t* is guaranteed to acquire all its locks since the criteria for execution requires all conflicting armored-transactions with a lower timestamp to finalize before *t* is executed. Notice that this does not undermine the irrevocability of the server's votes. Although the re-execution of a re-ordered armored-transaction creates a new vote, the new vote does not replace the vote from the first round; it becomes the vote of the ordering round.

Changing *t*'s timestamp may (a) have no effect on ℓ's locking queue order, for each data item ℓ accessed by *t*; (b) change the order between non-conflicting armored-transactions (e.g., when re-ordering a retrieve lock with other retrieve locks); or (c) change the order between conflicting armored-transactions (e.g., an update lock is re-ordered after a series of retrieve locks). In case (a), *ts* was increased but no conflicting transaction is affected and thus *t* keeps its position in the locking queue. In cases (b) and (c), *ts*'s increase resulted in a position change for *t* within ℓ's locking queue. In case (b), no transaction *u* with timestamp between *t*'s old timestamp and *ts* conflicts with *t* and so, the position of *t* in ℓ's locking queue is adjusted but no further actions are necessary. For case (c), the locking state of each conflicting transaction *u* has to be updated to assess whether *u* is eligible for re-execution (i.e., if *u* is the transaction with the lowest timestamp and has no conflicts).

Transaction termination. If the incoming vote certificate contains at least one partition vote for ABORT or COMMIT votes from all partitions, *s* treats the vote certificate as a *termination certificate*, i.e., a certificate that is

used to determine the *outcome* of t (Figure 4a, Step 7). If the outcome is ABORT, s rolls back t by discarding any update buffers; if COMMIT, s applies t's updates. In either case, t is no longer *pending* and its locks are released.

4.3 Execution with faulty clients

Faulty clients can attempt to disrupt the protocol by violating *safety* or *liveness* guarantees. The armored-transaction protocol builds on our prior work on Augustus [35] to ensure that safety cannot be violated. Callinicos does not provide absolute liveness guarantees. However, Callinicos does provide protection against some specific attacks. Below, we describe three in particular: (a) leaving a transaction unfinished in one or more partitions, which can also happen when a client fails by crashing; (b) forcing early termination of honest transactions (i.e., from a correct client); and (c) attempting denial-of-service attacks.

Unfinished transactions. A faulty client may leave a transaction unfinished by not executing all transaction rounds. We address this scenario by relying on subsequent correct clients to complete pending transactions left unfinished. If a transaction t conflicts with a pending transaction u in server $s \in g$, s will vote for ordering t. In the ORDER vote sent by s to the client, s includes u's operations. When the client receives an ORDER vote from $f_g + 1$ replicas in g, it forwards the ordering vote certificate for t and starts the termination of u by multicasting u's operations to every partition h involved in u. Clients do not have to immediately start the recovery of u, in particular when u is a multi-round transaction.

Once t is ordered, the client has a guarantee that t will eventually be executed. The amount of time that the client should wait before trying to recover u can be arbitrarily defined by the application. If a vote request for u was not previously delivered in h (e.g., not multicast by the client that created u), then the correct members of h will proceed according to the client's request. If u's vote request was delivered in h, then correct members will return the result of the previous vote, since they cannot change their vote (i.e., votes are final). If u is a single-round transaction, then the client will gather a vote certificate to finalize u. If u is a multi-round transaction, then the client will gather a vote certificate for the first execution round. In any case, eventually the client will gather enough votes to continue or finalize the execution of u, following the same steps as the failure-free cases.

Forced early termination of honest transactions. Faulty clients cannot force an erroneous or early termination of a honest transaction t. The atomic multicast protocol ensures that faulty clients cannot tamper with each others' transactions prior to delivery. Since we assume that faulty processes cannot subvert the cryptographic primitives, it is impossible for faulty clients to forge a transaction that will match the id of t once t is delivered. Furthermore, it is impossible for faulty clients to forge the vote certificates. Thus, once t is delivered to at least one partition, that partition will be able to enlist help from correct clients to disseminate t to the other partitions through the mechanism used to address unfinished transactions, and complete the execution of t.

Denial-of-service attacks. Faulty clients can try to perform denial-of-service attacks by submitting either: (a) transactions with many update operations, (b) multiple transactions concurrently, or (c) transactions with a very large number of rounds. Although we do not currently implement them in our prototype, a number of measures can be taken to mitigate such attacks, such as limiting the number of operations in a transaction, restricting the number of simultaneous pending transactions originating from a single client (e.g., [29]), or limiting the number of rounds in a transaction. These attacks, however, cannot force honest transactions to abort.

4.4 Correctness

In this section, we argue that for all executions H produced by Callinicos with committed update transactions and committed read-only transactions from correct clients, there is a serial history H_s with the same transactions that satisfies two properties: (a) If T reads an item that was most recently updated by T' in H (or "T reads from T'" in short), then T reads the same item from T' in H_s (i.e., H and H_s are equivalent). (b) If T commits before T' starts in H then T precedes T' in H_s.

Case 1. T and T' are single-partition transactions. If T and T' *access the same partition*, then from the protocol, one transaction executes before the other, according to the order they are delivered. If T executes first, T precedes T' in H_s, which trivially satisfies (b). It ensures (a) because it is impossible for T to read an item from T' since T' is executed after T terminates. If T and T' *access different partitions*, then neither T reads from T' nor T' reads from T, and T and T' can appear in H_s in any order to ensure (a). To guarantee (b), T precedes T' in H_s if and only if T commits before T' starts in H. In this case, recovering an unfinished transactions is never needed since atomic multicast ensures that T and T' are delivered and entirely executed by all correct servers in their partition.

Case 2. T and T' are multi-partition transactions that access partitions in PS (partition set) and PS', respectively.

First, assume that PS and PS' intersect and $p \in PS \cap PS'$. There are two possibilities: (i) either the operations requested by T and T' do not conflict, or (ii) at least one operation in each transaction conflict, and the transactions need to be ordered. In (i), property (a) is trivially ensured since neither transaction reads from the other,

otherwise they would conflict. To ensure (b), T and T' appear in H_s following their termination order, if they are not concurrent. If T and T' are concurrent, then their order in H_s does not matter. In (ii), from the algorithm (ii.a) T commits in every p before T' is executed at p, or (ii.b) T' commits in every p before T is executed at p, or (ii.c) T is executed first in a subset p_T of p and T' is executed first in the remaining (and complementary) subset $p_{T'}$ of p. For cases (ii.a) and (ii.b), without lack of generality, we assume (ii.a) holds. Thus, T precedes T' in H_s. Property (a) is guaranteed because it is impossible for T to read from T' since T will commit regardless of the outcome of T'. Property (b) holds because it impossible for T' to execute before T.

For case (ii.c), the vote certificate for T will contain COMMIT votes from p_T and ORDER votes from $p_{T'}$. Each of these votes contains a unique timestamp for T. The presence of an ORDER vote from a partition in PS forces all of PS to update the timestamp of T to the largest timestamp observed in the vote certificate (i.e., the final timestamp), and adjust the execution order of T according to this timestamp. If the final timestamp of T is smaller than the final timestamp of T', then T will be executed before T', and thus T precedes T' in H_s.

Now assume that PS and PS' do not intersect. Then T and T' can be in H_s in any order. In either case, (a) is trivially ensured. T precedes T' in H_s if and only if T commits before T' starts in H, and thus (b) is ensured. Recovery an unfinished transaction will extend its lifetime, but will not change the argument above.

Case 3. T is a single partition transaction that accesses partition P and T' is a multi-partition transaction that accesses partitions in PS'.

If T is executed before T' at P, T precedes T' in H_s. Property (a) follows from the fact that T cannot read from T'; property (b) follows because T' can only finish after T. If $P \notin PS'$, then (a) trivially holds and (b) can be ensured by placing T and T' in H_s following the order they complete. Finally, if T is executed after T' at P, then T' will precede T based on its final timestamp. If T' does not yet have a final timestamp, then T has to wait. If the final timestamp of T' remains smaller than the timestamp of T, then T' precedes T in H_s. Property (a) holds since T' cannot read from T and it is impossible for T to commit before T', as T has to wait for T'. Otherwise, T' will be such that T is executed before T', and thus T precedes T' in H_s as explained above.

5 Evaluation

Callinicos is designed to address three critical limitations of mini-transactions related to reliability, expressivity, and performance. These limitations manifest themselves in workloads that are *inherently unscalable*. Prior work on storage systems designed for cross-partition data-exchange or high contention workloads typically relax consistency requirements in order to meet performance demands (e.g., [48]). This observation is also reflected in the design of common storage system benchmarks. For example, widely used benchmarks like TPC-C [45] avoid queries that result in "hotspots". For this reason, we evaluate Callinicos using a set of micro-benchmarks. These micro-benchmarks are both inspired by real-world applications and illustrate the benefits of Callinicos on the types of workloads that cause most systems to perform poorly. Overall, our experiments demonstrate that Callinicos allows developers to easily build distributed applications with performance that scales with the number of partitions for high contention workloads.

Our prototype is implemented in Java 7. It includes an atomic multicast implementation based on PBFT [14] and the transaction processing engine. All source code is publicly available under an open source license.[1] In the experiments, each client is a thread performing synchronous calls to the partitions sequentially, without think time. Each partition contained four servers.

We ran all the tests on a cluster with the following configuration: (a) HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory, and (b) an HP ProCurve Switch 2910al-48G gigabit network switch. The single-hop latency between two machines connected to different switches is 0.17 ms for a 1KB packet. The nodes ran CentOS Linux 6.5 64-bit with kernel 2.6.32. We used the Sun Java SE Runtime 1.7.0_40 with the 64-Bit Server VM (build 24.0-b56).

5.1 Kassia distributed message queue

The first set of experiments measures the scalability of Callinicos, and evaluates the benefits of cross-partition communication. As a baseline, we compare the results against a mini-transaction implementation of the same queries. Although there are many examples of distributed data structures, including B-trees [1] and Heaps [11], we chose to focus on distributed queues because (i) the implementation of their basic operations are easy to explain, (ii) they are widely used by a number of real-world systems, including those at Yahoo, Twitter, Netflix, and LinkedIn [37], and (iii) they exhibit a natural point of contention, since every producer must read the location of the queue head when executing a *push* operation.

Our distributed message queue service, named Kassia, is inspired by Apache Kafka [5]. Both Kassia and Kafka implement a partitioned and replicated commit log service, which can be used to support the functionality of a messaging system. In both Kassia and Kafka, *producers*

[1] https://github.com/usi-systems/callinicos

generate new messages and publish them to *queues*, while *consumers* subscribe to queues. In both systems, queues can be assigned to different partitions. Kassia offers the same abstractions as Kafka, but with two crucial differences. First, while Kafka is only able to ensure a total ordering of messages on a single partition [6], Kassia (i.e., Callinicos) ensures a total ordering of messages across multiple partitions. Thus, Kassia, can distribute the load of message production and consumption by increasing the number of partitions. Second, while Kafka provides no reliability guarantees, Kassia tolerates Byzantine failures.

A message in Kassia is an entry of type (*msg_id*, *msg_content*), where *msg_id* is a key in the key-value store. A message queue is a sequence of pointers to *msg_id*'s. A designated storage area (i.e., the *head index*) is used to maintain an index that points to the first position in the queue. A second designated storage area (i.e., the *tail index*) is used to maintain an index that points to the last position in the queue.

To publish a message, a producer must perform the following operations: (i) read from the tail index to learn the location of the tail of the queue; (ii) increment the tail of the queue to the next location; (iii) write the message to the new tail location; and (iv) update the tail of the queue in the tail index.

Consumers in Kassia never remove messages from the queue. Instead they perform a sequential scan of all messages from the head to the tail. Thus, consumers perform the following operations: (i) read from the tail index to learn the location of the tail of the queue; (ii) read from the head index to learn the location of the head of the queue; (iii) issue a read range request to read all messages in the queue.

Thus, we see that producer operations exhibit high-contention, since all producer transactions include a write to the same location in the store. In contrast, consumer operations exhibit low-contention, read-only workloads.

We implemented the Kassia producer and consumer transactions using both armored-transactions and mini-transaction versions. Like the *swap* example from § 1, the mini-transaction version needed to be split into two separate transactions. The first transaction reads the tail index, while the second needs to use a compare operation to ensure that the value hasn't changed between the execution of the first and second transactions.

Scalability of Callinicos transactions. The first experiment evaluates how armored-transactions scale. We measured the maximum throughput for producer transactions as we increased the number of partitions. For this experiment, a separate queue was created in each partition and there were approximately six producers per queue. Thus, the workload itself is scalable. In other words, if all producers wrote to a single queue, we would not expect to see armored-transactions scale.

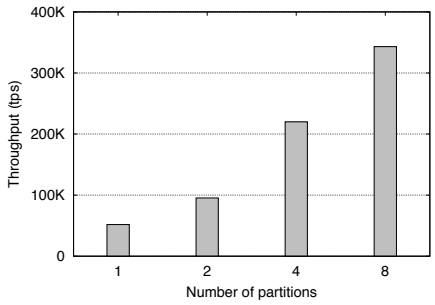


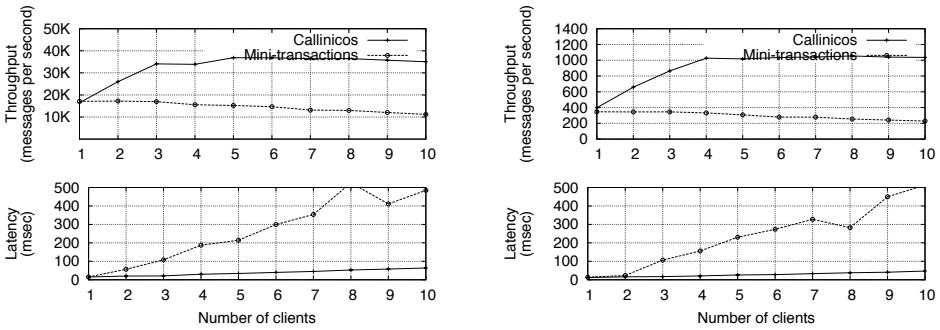
Figure 5: Maximum throughput for the producer (in transactions per second) with increasing number of partitions.

As shown in Figure 5, Callinicos scales nearly linearly with the number of partitions, achieving peak throughput of 60k, 116k, 220k and 350k messages per second with 1, 2, 4 and 8 partitions, respectively. In this experiment, producers submit batches with 200 messages of 200 bytes.

Armored-transactions vs. mini-transactions. The next experiment considers the question of how armored-transactions perform compared to mini-transactions for *high-contention* and *low-contention* workloads. For high-contention workloads, an initially empty queue was repeatedly written to by increasing number of producers. For low-contention workloads, an initially full queue was repeatedly read by increasing numbers of consumers. Recall that consumers do not remove messages from the queue; each consumer operation reads all the messages in the queue. For both workloads, we measured the throughput and latency as we increased load.

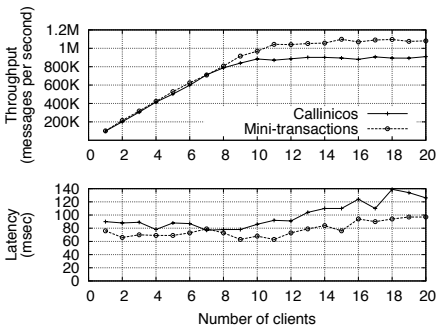
For this experiment, there was a single queue distributed across 4 partitions. Thus, in contrast to the scalability measurements above, the workload is inherently unscalable. As a further parameter to the experiment, we varied the number and size of messages sent by the clients. Recall that each client executes a single thread that repeatedly sends batches of synchronous messages. In the first configuration, clients send 200 messages of 200 bytes in each batch. In the second configuration, they send 4 10-KB messages in each batch.

Figures 6 (a) and (b) show the producer results. With 200-byte messages, Callinicos outperforms mini-transactions at peak throughput by a factor of 2.1. Moreover, due to the high number of aborts experienced by mini-transactions, producers need to resubmit their requests, which increases latency. With 10k-byte messages, Callinicos outperforms mini-transactions by a factor of 2.5. Both systems present similar latency up to 2 clients, but the latency of mini-transactions increases quickly with larger number of clients. In both configurations, we can

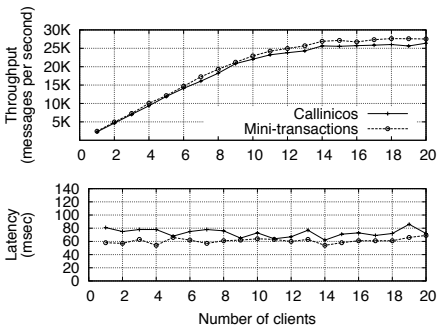


(a) 200-byte messages (producer)

(b) 10k-byte messages (producer)



(c) 200-byte messages (consumer)



(d) 10k-byte messages (consumer)

Figure 6: Compared to mini-transactions, Kassia producers demonstrate higher throughput and lower latency compared to mini-transactions for contention-heavy workload. Kassia consumers show no overhead for contention-light workload.

also observe that the two system behave differently once they reach their point of saturation: armored-transactions experience quite stable performance, while in the case of mini-transactions performance drops.

Figures 6 (c) and (d) show the consumer results. In the absence of contention, mini-transactions perform slightly better than Callinicos armored-transactions with small messages; with larger messages the difference between the two systems is negligible. In both cases, mini-transactions consistently display lower latency than armored-transactions, although as for throughput, the difference is more noticeable with small messages. Since the consumer workload is read-only, there are no aborts and both systems sustain throughput at high load.

Our experiments show that Callinicos provides better throughput and latency than mini-transactions for high-contention workloads. For low-contention workloads, Callinicos adds no additional overhead.

5.2 Buzzer distributed graph store

As a second application, we implemented a Twitter clone named Buzzer, which is backed by a distributed graph store. Most social networking applications exhibit only eventually-consistent semantics [48]. As a result, users have become accustomed to odd or anomalous behavior (e.g., *one friend sees a post, and another doesn't*). In contrast, Buzzer not only provides strict serializability, but also tolerates Byzantine failures. Moreover, graph stores offer a useful point-of-comparison because: (i) they demonstrate a different design from queues where contention is less pronounced (i.e., all writers don't update the same memory location), and (ii) graphs have become increasingly popular for applications in telecommunications, transportation, and social media [42].

In Buzzer, there are operations to *post* a message, *follow* a user (friends), and to retrieve one's *timeline* (i.e., an aggregate of the friends' posts). *Posts* are single-round, single-partition update transactions. *Follows* are single-round, multi-partition update transactions. *Timelines* are multi-round, multi-partition read-only transactions.

The level of contention in a graph store depends on the workload and the structure of the graph on which the workload is run. For these experiments, we used a single workload that was composed of 85% *timeline*, 7.5% *follow*, and 7.5% *post* operations, and varied the

contention by altering the connectivity of the graph. For typical graph queries that update data as they traverse the graph, dense, highly connected graphs exhibit high contention, while sparse graphs exhibit low contention. We opted for a small social network with 10,000 users. Then, using statistics from Twitter [46], we inferred that the "friending" behavior of Twitter users approximately follows a Zipf distribution with size 2,500 and skew of 1.25. We built our social network graph by sampling that Zipf distribution. We call this network *high-contention*. To contrast with this highly connected network, we created another network using a Zipf distribution with size 25 and skew 1.25. We call this network *low-contention*.

In these experiments, we explored the question of *how Callinicos' conflict resolution compares to optimistic concurrency control*. The experiments measure the maximum throughput for the workload described above.

While partitioning the queue data was relatively straightforward, ensuring a good partitioning of the social network data is somewhat more complex. We distributed relationships such that each user has a 50% probability of having her friends' data on the same partition where her data is. If not on the same partition, all friends' data is placed on two partitions, the user's partition and another partition chosen randomly. For example, for the four-partition scenario all data accessed by a user has a 50% chance of being fully contained in a single partition and a 50% chance of being spread across two different partitions (16.67% for each combination). We enforced this restriction to assess the scalability of the system as the number of partitions increase.

The results, seen in Figure 7, show that Callinicos' conflict management adds little overhead for low-contention workloads. However, for high-contention workloads, Callinicos' demonstrates significantly better throughput. In other words, although ordering conflicting transactions adds a slight overhead, it avoids aborts which are even more detrimental to performance. Moreover, these experiments show that for both high-contention and low-contention workloads, the throughput of Callinicos scales with the number of available partitions.

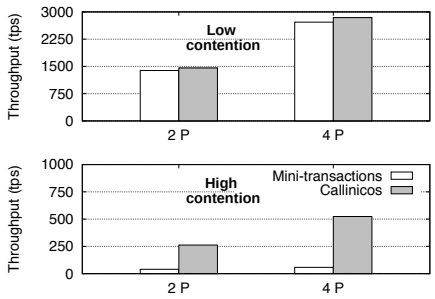


Figure 7: Buzzer's maximum throughput under low- and high-contention for 2 and 4 partitions (2 P and 4 P).

6 Related work

Many storage systems have been designed and implemented. In this section, we compare Callinicos to many of these systems from a few different perspectives.

Distributed storage systems. Database systems that implement some notion of strong consistency (e.g., serializability, snapshot isolation) traditionally use two-phase locking (2PL), optimistic concurrency control (OCC), or a variation of the two to coordinate the execution of transactions. Examples of distributed database systems based on 2PL are Gamma [20], Bubba [13], and R* [32]. Spanner [17] uses 2PL for update transactions and a timestamp-based protocol for read-only transactions. Examples of recent systems based on OCC are H-Store [25] and VoltDB [49]. MDCC [27] and Geo-DUR [40, 41] use OCC for geo-replicated storage. Percolator implements snapshot isolation using OCC [12]. In a seminal paper, Gray et al. [24] have shown that two-phase locking and optimistic concurrency control are exposed to a prohibitively high number of deadlocks and aborts when used to handle replication. Fundamentally, the problem stems from the fact that requests are not ordered among replicas, a limitation that is addressed by Callinicos.

Storage systems with limited transactions. Several distributed storage systems improve performance by supporting limited types of distributed transactions. For example, MegaStore [7] only provides serializable transactions within a data partition. Other systems, such as Granola [18], Calvin [44] and Sinfonia [2] propose concurrency control protocols for transactions with read/write keys that are known a priori. Moreover, many cloud-oriented storage systems have abandoned transactional properties to improve performance. For example, Apache Cassandra [3], Apache CouchDB [4], MongoDB [33], and Amazon Dynamo [19] offer no transaction support. When using such systems, applications must handle contention explicitly, by preventing conflicts from happening or by allowing weaker consistency (e.g., eventual consistency). NoSQL scalable storage systems that offer transactions are usually limited to single-row updates (e.g., [15]) or single-round transactions (e.g., [2, 18]), and employ optimistic concurrency control (e.g., [2, 7, 35]).

Storage systems with support for contention. A few systems have been proposed that can handle contention, although none of these systems tolerate Byzantine failures. Calvin [44] deals with contention by preemptively ordering all transactions. It adds a sequencing layer on top of any partitioned CRUD storage system and enables full ACID transactions. The sequencing layer operates in rounds. Each round lasts 10 ms, which is used to batch

incoming requests. Once a sequencer finishes gathering requests for a given round, it exchanges messages with other sequencers to merge their outputs. The sequencing layer provides ordering without requiring locks, and supports transactional execution without a commit round. Calvin, however, does not support multi-round execution.

Rococo [34] uses a dependency graph to order transactions. Transactions are broken into pieces that can exchange data. Pieces are either immediate or deferrable. An offline checker analyzes all transactions in the system and decides which can be reordered based on their pieces. Once executed, an immediate piece cannot be reordered, i.e., transactions with conflicting immediate pieces cannot be reordered. A central coordinator distributes pieces for execution, forwards intermediary results, and collects dependency information. At commit, each server uses the dependency graph information to reorder deferrable pieces. Callinicos can reorder all types of transactions and does it using a simpler algorithm based on timestamps.

H-Store [43] promotes the idea of favoring single-partition transactions executed in a single thread without any contention management. This approach ensures optimal performance under no contention. When the number of aborts increases (i.e., under heavier contention), H-Store first tries to spread conflicting transactions by introducing waits, and then switches to a strategy that keeps track of read and write sets to try to reduce aborts. Callinicos keeps track of the read and write sets by default, and orders and enqueues conflicting transactions instead of aborting them. Multi-round execution in H-Store depends on a centralized coordinator, which is responsible for breaking transactions into *subplans*, submitting these subplans for execution, and executing application code to determine how to continue the transaction.

A speculative execution model on top of H-Store is proposed in [25], where three methods of dealing with contention are compared. The first method, *blocking*, simply queues transactions regardless of conflict. The second method, *locking*, acquires read and write locks, and suspends conflicting transactions. If a deadlock is detected, the transaction is aborted. The third method is *speculative execution*. Conflicting multi-partition transactions are executed in each partition as if there were no contention, and if they abort due to contention their rollback procedure requires first rolling back all subsequent transactions and then re-executing them. Multi-round execution follows the model proposed by H-Store. Experimental evaluation of the speculative model shows that for workloads that incur more than 50% of multi-partition transactions, the throughput of the speculative approach drops below the locking approach. In a multi-round transaction benchmark, the throughput of the speculative model dropped to the level of the blocking approach.

Granola [18] uses timestamps to order transactions. The timestamps are used in two different types of transactions: *independent* and *coordinated*. For independent transactions, replicas exchange proposed timestamps, select the highest proposal, and execute the transaction at the assigned timestamp. Coordinated transactions also exchange timestamp proposals to order transactions, but they abort if any replica votes ABORT or detects a conflict.

Byzantine fault-tolerant storage systems. Some storage systems have been proposed that can tolerate Byzantine failures. MITRA [30, 31] and Byzantium [22] are middleware-based systems for off-the-shelf database replication. While MITRA implements serializability, Byzantium provides snapshot isolation. BFT-DUR [36] and Augustus [35] are transactional key-value stores. BFT-DUR is an instance of deferred update replication and can support arbitrary transactions. Augustus provides an interface similar to Sinfonia, with transactions with operations defined a priori. All these systems rely on optimistic concurrency control to ensure strong consistency. Thus, in the case of contention they are exposed to many aborts. HRDB [47] provides BFT database replication by relying on a trusted node to coordinate the replicas. Although HRDB provides good performance, the coordinator is a single point of failure. DepSky [9] provides confidentiality for cloud environments. It offers users a key-value interface without the abstraction of transactions.

7 Conclusion

Callinicos introduces a generalized version of the single-round mini-transaction model. Multi-round armored-transactions are modeled using a transaction matrix, in which rows indicate the rounds, columns indicate the partitions, and each cell contains the commands that a partition will execute during a round. Callinicos allows for cross-partition communication, and, uses a timestamp-based approach to reorder and ensure the execution of conflicting transactions. Using Callinicos, we've implemented Kassia, a distributed BFT message queue, and Buzzer, a social network backed by a distributed graph store. that scales with the number of partitions. Overall, our experiments show that Callinicos offers scalable performance for high-contention workloads, and that the design allows users to implement robust, performant distributed data structures.

Acknowledgements

We wish to thank Benjamin Reed, our shepherd, and the anonymous reviewers for the constructive comments. This work was supported in part by Microsoft Research through its PhD Scholarship Programme and the Hasler Foundation.

References

[1] AGUILERA, M. K., ET AL. A practical scalable distributed b-tree. *Proc. VLDB Endow. 1*, 1 (2008), 598–609.

[2] AGUILERA, M. K., ET AL. Sinfonia: A new paradigm for building scalable distributed systems. *TOCS 27*, 3 (2009), 5:1–5:48.

[3] Apache Cassandra. `http://cassandra.apache.org/`.

[4] Apache CouchDB. `http://couchdb.apache.org/`.

[5] Apache Kafka. `http://kafka.apache.org/`.

[6] Kafka 0.9.0 Documentation. `https://kafka.apache.org/documentation.html`.

[7] BAKER, J., ET AL. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011).

[8] BARTLETT, W., AND SPAINHOWER, L. Commercial fault tolerance: A tale of two systems. *IEEE TDSC 1*, 1 (Jan. 2004), 87–96.

[9] BESSANI, A., ET AL. Depsky: Dependable and secure storage in a cloud-of-clouds. In *EuroSys* (2011).

[10] BESSANI, A., ET AL. State machine replication for the masses with bft-smart. In *DSN* (2014).

[11] BHAGWAN, R., ET AL. Cone: A distributed heap-based approach to resource selection. Tech. rep., UCSD, December 2004.

[12] BHATOTIA, P., ET AL. Large-scale incremental data processing with change propagation. In *HotCloud* (2011).

[13] BORAL, H., ET AL. Prototyping bubba, a highly parallel database system. *IEEE TKDE 2*, 1 (Mar. 1990), 4–24.

[14] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance and proactive recovery. *TOCS 20*, 4 (November 2002), 398–461.

[15] CHANG, F., ET AL. Bigtable: A distributed storage system for structured data. In *OSDI* (2006).

[16] CLEMENT, A., ET AL. BFT: The time is now. In *LADIS* (2008), pp. 1–4.

[17] CORBETT, J. C., ET AL. Spanner: Google's globally distributed database. *ACM TOCS 31*, 3 (Aug. 2013), 8:1–8:22.

[18] COWLING, J., AND LISKOV, B. Granola: Low-overhead distributed transaction coordination. In *USENIX ATC* (2012).

[19] DECANDIA, G., ET AL. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[20] DEWITT, D. J., ET AL. The gamma database machine project. *IEEE TKDE 2*, 1 (Mar. 1990), 44–62.

[21] DWORK, C., ET AL. Consensus in the presence of partial synchrony. *JACM 35*, 2 (April 1988), 288–323.

[22] GARCIA, R., ET AL. Efficient middleware for byzantine fault tolerant database replication. In *EuroSys* (2011).

[23] GRAY, J. A census of tandem system availability between 1985 and 1990. *IEEE TOR 39*, 4 (1990), 409–418.

[24] GRAY, J., ET AL. The dangers of replication and a solution. *SIGMOD Rec. 25*, 2 (June 1996), 173–182.

[25] JONES, E. P., ET AL. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD* (2010).

[26] KOTLA, R., ET AL. Zyzzyva: Speculative byzantine fault tolerance. *ACM TOCS 27*, 4 (2010), 1–39.

[27] KRASKA, T., ET AL. Mdcc: Multi-data center consistency. In *Eurosys* (2013).

[28] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.

[29] LISKOV, B., AND RODRIGUES, R. Tolerating byzantine faulty clients in a quorum system. In *ICDCS* (2006).

[30] LUIZ, A. F., ET AL. Byzantine fault-tolerant transaction processing for replicated databases. In *NCA* (2011).

[31] LUIZ, A. F., ET AL. Mitra: Byzantine fault-tolerant middleware for transaction processing on replicated databases. *SIGMOD Rec. 43*, 1 (May 2014), 32–38.

[32] MOHAN, C., ET AL. Transaction management in the R* distributed database management system. *ACM TODS 11*, 4 (Dec. 1986), 378–396.

[33] MongoDB. `http://www.mongodb.org/`.

[34] MU, S., ET AL. Extracting more concurrency from distributed transactions. In *OSDI* (2014).

[35] PADILHA, R., AND PEDONE, F. Augustus: Scalable and robust storage for cloud applications. In *Eurosys* (2013).

[36] PEDONE, F., AND SCHIPER, N. Byzantine fault-tolerant deferred update replication. *Journal of the Brazilian Computer Society 18*, 1 (2012), 3–18.

[37] Powered By Apache Kafka. `https://cwiki.apache.org/confluence/display/KAFKA/Powered+By`.

[38] PRABHAKARAN, V., ET AL. Iron file systems. In *SOSP* (2005), pp. 206–220.

[39] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM CSUR 22*, 4 (1990), 299–319.

[40] SCIASCIA, D., ET AL. Scalable deferred update replication. In *DSN* (2012).

[41] SCIASCIA, D., AND PEDONE, F. Geo-replicated storage with scalable deferred update replication. In *DSN* (2013).

[42] SOULÉ, R., AND GEDIK, B. Railwaydb: adaptive storage of interaction graphs. *The VLDB Journal* (2015), 1–19.

[43] STONEBRAKER, M., ET AL. The end of an architectural era (it's time for a complete rewrite). In *SIGMOD* (2007).

[44] THOMSON, A., ET AL. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012).

[45] TPC-C. `http://www.tpc.org/tpcc/`.

[46] Twitter Statistics. `http://www.beevolve.com/twitter-statistics/`, Oct. 2012.

[47] VANDIVER, B., ET AL. Tolerating Byzantine Faults in Transaction Processing Systems Using Commit Barrier Scheduling. In *SOSP* (2007).

[48] VENKATARAMANI, V., ET AL. Tao: How facebook serves the social graph. In *SIGMOD* (2012).

[49] VoltDB. `http://www.voltdb.com/`.

[50] YANG, J., ET AL. Explode: A lightweight, general system for finding serious storage system errors. In *OSDI* (2006).

Filo: consolidated consensus as a cloud service

Parisa Jalili Marandi Christos Gkantsidis Flavio Junqueira Dushyanth Narayanan
Microsoft Research, Cambridge, UK

Abstract

Consensus is at the core of many production-grade distributed systems. Given the prevalence of these systems, it is important to offer consensus as a cloud service. To match the multi-tenant requirements of the cloud, consensus as a service must provide performance guarantees, and prevent aggressive tenants from disrupting the others. Fulfilling this goal is not trivial without over-provisioning and under-utilizing resources.

We present Filo, the first system to provide consensus as a multi-tenant cloud service with throughput guarantees and efficient utilization of cloud resources. Tenants request an SLA by specifying their target throughput and degree of fault-tolerance. Filo then efficiently consolidates tenants on a shared set of servers using a novel *placement algorithm* that respects constraints imposed by the consensus problem. To respond to the load variations at runtime, Filo proposes a novel *distributed controller* that piggybacks on the consensus protocol to coordinate resource allocations across the servers and distribute the unused capacity fairly. Using a real testbed and simulations, we show that our placement algorithm is efficient at consolidating tenants, and while obtaining comparable efficiency and fairness, our distributed controller is $\sim 5x$ faster than the centralized baseline approach.

1 Introduction

Distributed consensus is a problem where several processes must agree on a common value [11]. Consensus is used by many systems to elect leaders [43, 38, 31], access shared objects [9], and order transactions in replicated services [44, 30, 8]. Consensus protocols are complex and difficult to implement correctly. Because of this difficulty, systems such as ISIS [7], Chubby [9], Apache ZooKeeper [23], and OpenReplica [46] offer consensus as a black box service.

Why consensus as a cloud service. Consensus is at the heart of many distributed applications that are or will become cloud residents. Hence, offering consensus as a *simple-to-use cloud service* will relieve people from the hurdles of developing and maintaining it. Similarly to systems such as key-value stores that are ubiquitously available as cloud services [13, 10, 3], we envision a future where consensus is available as a cloud offering.

Why consolidated consensus. The key to a cloud provider's success is reducing server footprint; this means reduced costs for the provider and also for the users. *Sharing* computing, storage, and network resources is essential for decreasing the number of servers. Our goal then is to consolidate consensus instances of many tenants (users) and deploy them on shared servers. Sharing is advantageous: reserved but under-utilized resources can be (temporarily) re-allocated to other tenants who demand higher performance at runtime, and prices can be accordingly adjusted. Re-allocations are impossible if tenants are promised dedicated resources. When sharing, it is essential to dynamically adjust allocations at real time to minimize the durations for which resources remain idle, and to use techniques that prevent aggressive tenants from monopolizing the resources.

Filo. Thus, to successfully offer consensus as a cloud service we should (a) design a simple API, (b) efficiently and fairly utilize shared servers, and (c) ensure performance isolation among tenants. We propose Filo, a multi-tenant consensus service that provides throughput SLA guarantees and meets these requirements. Filo's API is intuitive: users specify their SLAs in application-level request rates, and choose from a range of reliability options by specifying the number of replicas and the durability mode of their data (*i.e.*, storing data in memory, or on stable storage), and thus trade performance or price for reliability.

Filo co-locates replicas of various tenants on shared servers (consolidation). To consolidate, we have designed a novel placement algorithm that packs replicas,

while respecting SLAs and the constraints imposed by the properties of the consensus protocols (*e.g.*, replicas of a tenant are placed on distinct servers to resist failure). Filo uses an empirical approach similar to [26, 1] to translate the high level SLAs specified by the users to low level resource costs (*i.e.*, CPU, network bandwidth, storage IO). Our placement algorithm uses these values to select servers that will host the replicas.

Consolidating replicas on shared servers raises an immediate concern with performance isolation. Filo guarantees performance isolation by rate limiting requests using multi-resource token buckets [1]. Filo dynamically calculates and tunes the budgets of these token buckets.

Anecdotal evidence suggests that tenants may misestimate their SLA requirements. To account for these inaccuracies, Filo monitors resource utilization at runtime and fairly re-distributes free resources among the tenants using a distributed controller. Centralized controllers are popular due to their accuracy at efficient resource allocations. However, the overhead of these techniques is also well-understood: all the servers in a cluster must transfer their status to the centralized controller regularly, and computations must be performed over a rather large set of data. Our contribution here is the design of a *distributed controller* that eliminates this overhead and yet results in highly efficient resource allocations. In a system such as Filo, where a tenant's replicas are placed on a rather small subset of servers (*i.e.*, 3, 5), a distributed controller has a high opportunity for calculating allocations that compete closely with the centralized techniques. To coordinate resource allocations across the servers, our distributed controller leverages the underlying consensus connections to transfer its messages.

Contributions. To summarize, in addition to building consolidated consensus as a cloud service, this paper makes the following contributions: (i) we propose an API that simplifies user's interactions with the system and abstracts away low level resource specifications that are needed for efficient replica placement, (ii) we design a novel placement algorithm to efficiently pack replicas on a shared set of servers while respecting the constraints imposed by consensus protocols, and (iii) we design a new *distributed controller* that dynamically, fairly, and efficiently adjusts resource allocations at runtime.

Results. We have implemented consolidated consensus and evaluated its performance in various workloads. These evaluations are essential for bounding feasible SLAs and translating SLAs to resource costs. We have thoroughly evaluated our placement algorithm and the distributed controller. Our placement algorithm efficiently places replicas on the servers while allocating above 98% of the resources. And our distributed controller converges to allocations with 95% efficiency of the centralized techniques while being $\sim 5x$ faster.

2 Consensus in Filo

Consensus enables a set of processes (*replicas*) to reach agreement on some value [11]. Consecutive execution of consensus produces a *log* of *ordered* entries. Each entry indicates the request that must be executed next in the sequence by the application. Filo provides consensus as a service by assigning to each tenant a set of replicas. We refer to each such set that executes consensus and orders the requests as a *consensus group*, and to the size of this group as *replication degree*. To be fault tolerant, replicas of a tenant must be placed on distinct servers. We differentiate between a *server* and a *replica*. A server is a physical machine and can host multiple replicas.

Separating ordering from execution. Tenants use Filo only for the *ordering and durability* guarantees. It is left to the tenants to *execute* their ordered requests, as Filo is agnostic to user-application semantics and treats requests as arbitrary bytes. Separating ordering from the execution has been the subject of previous studies [51, 8, 28]. This design choice is of crucial importance to Filo as it targets a diverse collection of cloud residents each with their own specific applications. Thus, we differentiate between the servers on which tenant applications are running (*execution*) and the servers on which Filo is running (*ordering*). We assign a *dedicated cluster of servers* to Filo which is not used by other applications. [1] Thus since tenant applications are outside the boundaries of Filo, they are free to use their desired caching or replication mechanisms, while using Filo only for the ordering and durability of their requests.

Consolidated consensus using Chain Replication. Filo implements consensus using Chain Replication [42]. Each tenant is assigned a chain composed of n replicas to which we will refer as *tenant replicas*. These are deployed on Filo's server cluster. Chain Replication organizes replicas on a chain and distinguishes between *head* and *tail* replicas. Clients send their write and read requests to the head and tail replicas respectively; thus both of these replicas communicate with the clients. To simplify the management of the client connections, and without affecting correctness, here we replace chain with a ring. The tail sends an ack to the head replica when a write request is finalized. Read and write requests are all received by the head. We refer to the replicas other than the head as *followers*. Note that we do not dedicate one server to each tenant replica, but rather servers are shared among the replicas belonging to various chains. We refer to this as the *consolidated* deployment of the chains.

Why Chain Replication. Majority-based protocols such as Paxos [30] need $2f+1$ replicas to tolerate f fail-

[1] Dedicating a cluster to a critical service such as Filo is important to protect its performance.

ures. Higher number of replicas enhances liveness but not performance. Given the importance of resource efficiency, we prefer protocols that achieve similar performance with fewer replicas. Chain Replication is one such protocol that with $f+1$ replicas tolerates f failures. But unlike Paxos, to make progress it requires *all* the replicas to be non-faulty. Because of its efficiency we have chosen Chain Replication to implement consensus. Despite this the choice of a consensus protocol is orthogonal to the focus of this paper and the techniques proposed in this paper are generalizable to others.

Failures. When replicas in a chain fail, the chain must be reconfigured before it can resume its operation with the initial degree of resiliency. As in [42], here failures are detected and resolved by an external replicated *master*. In our consolidated model there is only one master that is shared by all the chain instances. This is similar to Vertical Paxos [32] where many groups exists in the system and a master handles reconfigurations for all of them (see § 4.1 for more details).

Assumptions. We assume a crash-recovery failure model and exclude byzantine failures. Network is asynchronous with a possibility of message loss and arbitrary delays. We deploy consensus groups in *in-mem*, or *persistent* durability modes. In the latter, *logs* are made durable on a storage device. The system needs $f+1$ replicas to tolerate f failures.

Logger. To manage the *logs*, we have designed a subsystem similar to Bookkeeper [2] called *Logger*. *Logger* implements *group commit* with a buffer for receiving requests. As the number of concurrent requests increases, buffers transmit more data to the storage device with every fluch, and, hence, efficiency increases (without being subject to timeouts). Compared with *random* writes, *Sequential* writes benefit the most from group commit; thus we have one sequential log per server to which requests from all tenants are appended. *Logger* constantly and in the background prepares per-replica logs on a second storage device. Our servers must be provisioned by at least two storage devices for better performance.

3 System Design

Filo's main components are: (a) a simple API that allows tenants to specify their performance and reliability requirements, and (b) rate limiters and the resource allocator that manage server resources.

API. A new applicant starts by submitting an *admission request* to the *Admission Controller* (*AC*), a central component of the system. An admission request contains the following attributes:

$$(durability\ mode, replication\ degree, request\ size, throughput\ SLA) \quad (1)$$

Durability mode is either *in-mem* or *persistent*; replication degree is the number of requested servers; the throughput SLA is the application-level request rate for requests of a fixed size (request size). (For brevity, we omit attributes related to user account and other authentication information.) If the admission request can be satisfied by Filo the applicant is admitted to the system, after which it becomes a tenant. The tenant is given a handle to its consensus group, which will be used for submitting requests. Filo guarantees rates up to the throughput SLA, and can also accommodate higher rates if there is available capacity (see § 4.2 for more details).[2] The tenants submit read or write requests. Writes trigger consensus and are appended to the *log* (see § 2). Reads retrieve previously ordered data from the log. The throughput SLA agreed above is the sum of read and write requests. Writes are more expensive than reads as they are propagated to other replicas. Hence we concentrate on the write requests.

Rate limiters. To enforce throughput SLAs, Filo installs rate limiters in the form of multi-resource token buckets [49, 1] on the servers that host head replicas of the consensus groups and also on the external servers on which tenant applications execute. The token budget for each tenant is determined based on the request size specified in its *admission request* (1). Note that tenants are not prohibited from varying their request size at runtime. SLAs apply only to the initial request sizes, however. To prevent this variation from affecting other tenants, a request passes through the system if token buckets have sufficient tokens; requests of different sizes translate to different amount of tokens. Moreover, Filo constantly monitors resource usage at runtime (see below) and adjusts the budgets dynamically.

Resource allocator. The most important element of Filo is the resource allocator which is composed of two entities: an admission controller that executes the admission phase, and a distributed controller that executes the work conservation phase. During the admission phase, admission controller runs a placement algorithm to efficiently place tenant replicas on the servers. Placement is done based on the translation of the admission requests to resource usages (CPU, network bandwidth, storage IO), which is done using an empirical strategy; prior to

[2] Before launching Filo, we extensively evaluate its performance under various workloads to find its peak performance. Peak performance caps the range of SLAs that Filo can promise to its users. For example if we can order 10 reqs/sec at best, we can not admit an applicant that asks for 11 reqs/sec.

launching the service, we extensively benchmark the system to build a performance profile. During the work conservation phase, the distributed controller monitors resource utilization and adjusts the allocations to absorb the exceeding demands of its tenants.

4 Resource Allocator

Filo's objective is to offer consensus as a service while providing SLA guarantees, and fairly and efficiently allocating CPU, network, and storage resources. To accomplish these goals, it is essential to manage replica placement when admitting tenants, to dynamically adjust resource allocations at runtime, and to control the rates at which tenants submit their requests. *We have designed resource allocator to handle these tasks*. We next define the concepts and constraints of our problem, and then present our algorithms for implementing the *resource allocator* via the admission and work conservation phases.

Definitions. Let S be a set of m servers, each with k resources. Let $(r_1,...,r_k)$ be a resource vector. Each server has two vectors R^{no} and R for the nominal and free amount of its resources ($r \in \mathbb{R}_{\geq 0}$, and R^{no} is constant). Let A and T be the set of applicants and tenants. An applicant turns to a tenant if it is admitted to the system. Let n_t be the replication degree of tenant t. Tenant t has a *demand profile* P, which is a set of n_t *demand vectors* (one vector per replica). Demand vector p specifies the resources needed by one replica. Let F_a, *feasibility region* of applicant a, be the set of servers that can be considered for placing its replicas. Let E_t be the set of the elected servers at the end of t's admission ($|E_t| = n_t$).

Example setup. To simplify the description of our algorithms in the next sections, we outline a hypothetical example setup here. Assume 4 servers and 3 resources as CPU (count), network bandwidth (Gbps), and storage IO (IOps). Resource vectors of our servers are $R_1 = (2,2,250)$, $R_2 = (2,2,400)$, $R_3 = (8,4,100)$, $R_4 = (4,2,250)$. We assume 2 applicants with 2 replicas each, and resource profiles as shown in Table 1.[3]

Constraints. *Resource allocator* is subject to the following constraints:

- **Cons$_1$.** A replica is *indivisible*; all the values in its vector are demanded from exactly *one* server, and it is *placed* if there exists a server to satisfy it.
- **Cons$_2$.** An applicant is *indivisible*; an applicant is *admitted* if there is a set of servers that can host *all* of its replicas.
- **Cons$_3$**. To cope with failures a server can host at most one replica of an applicant.

[3] In Chain Replication the head replica is loaded more than the other roles. Similarly in Paxos protocol, the replica that plays the coordinator role demands more resources than the others.

Applicant	Replica	CPU	Net-bw	Storage IO
a_1	p_1	1	1	100
	p_2	3	1	100
	p_1+p_2	**4**	**2**	**200**
a_2	p_1	3	④	100
	p_2	1	1	150
	p_1+p_2	**4**	**5**	**250**
$R_1+R_2+R_3+R_4$		**16**	**10**	**1000**

Table 1: Example setup

Note. Algorithms designed in this section are inspired by DRF [20] due to its fairness criteria: when dividing multiple goods among many suppliants the goal is to maximize each suppliant' share, while equalizing the share of their most demanded good. Moreover, DRF is computationally light which is essential for making quick resource adjustments (see § 7).

4.1 Admission Phase

Admission phase is executed by the admission controller (*AC*), which analyzes *admission requests* and finds servers to place replicas. We propose a *placement algorithm* that efficiently places replicas on shared servers. Our algorithm runs until all applicants are admitted to the system or denied admission if satisfying their SLAs is infeasible. As described in § 3 an applicant uses Filo's API to specify its desired SLA and reliability requirements, which are then translated to CPU, network, and storage usage and saved as its demand profile. Using these profiles at each iteration the algorithm must decide: (a) which *applicant*, (b) which *replica* of the selected applicant, and (c) which *server*, to consider next in its allocations so to be efficient. Our choices are based on 3 policies that we explain next with the rational for each:

(PO$_a$). To maximize resource usage, we prioritize the applicant with the highest *dominant share*. To get an applicant's dominant share: we divide the aggregated resource demand of its replicas by the total free amount of resources in the system, and choose the highest value (see Eq 2). In the above example we prioritize applicant a_2 with the dominant share of $5/10$ (see Table 1).

(PO$_b$). For the selected applicant, we prioritize the replica with the highest dominant share. If there is no server to fit the heaviest replica, there is no point in fitting the others. To get a replica's dominant share: we divide its demand by the total free amount of the resources in the system, and choose the highest value among the resources. In the above example we prioritize a_2's first replica with the dominant share of $4/10$ (due to its network bandwidth, note the circle in Table 1).

(PO$_c$). To place the selected replica, we prioritize the servers with more free resources. The rational is to eventually balance resource utilization across the servers.

Placement Algorithm. Alg. 1 encapsulates our constraints and policies in more detail. The dominant share of applicant a (line 2) is obtained as follows. Before the admission, the feasibility region of an applicant is the entire server cluster, hence, its dominant share must be calculated without constraining the server set. Thus we compute the global amount of free resources as $R^g = \sum_{\forall s \in S} R_s$. We use a similar formula to calculate ${P_a}^g$ for applicant a (in the example $R^g = (16, 10, 1000)$, ${P_{a_1}}^g = (4, 2, 200)$, and ${P_{a_2}}^g = (4, 5, 250)$, as in Table 1). Then we use Eq 2 to calculate a's dominant share:

$$D_a = \max_{i=1}^{k} \{({P_a}^g)_i / (R^g)_i\} \quad (2)$$

(*e.g.*, D_{a_2} is $5/10$). R^g and ${P_a}^g$ remain constant during the execution of the algorithm. Thus, the dominant shares of the applicants are calculated once at the beginning. Similarly, D_p, dominant share of replica p (line 5) is calculated using Eq 2, where p_i replaces $({P_a}^g)_i$. To find the most free servers we calculate servers' dominant shares (line 20) relative to their nominal capacity using Eq 3:

$$D_s = \max_{i=1}^{k} \{(R_s^{no} - R_s)_i / (R_s^{no})_i\} \quad (3)$$

i.e., dominant share of a server is determined by its most consumed resource. As R_s is modified frequently, at the end of an admission, D_s is recalculated to respect $\mathbf{PO}_c$ (line 15). Updating D_s at the middle of an admission is unnecessary as the elected servers are removed from the feasibility region to respect $\mathbf{Cons}_3$ (line 24). Resource usage across the servers is gradually equalized ($\mathbf{PO}_c$) using *ElectServer*. Once an applicant is admitted (*i.e.*, all of its replicas are placed), Filo activates token buckets and assigns their budgets based on the agreed SLA.

Failures. A chain is disrupted if any of its replicas fails. We invoke *ElectServer* on demand to replace failed replicas. Replacing failed replicas is prioritized over admitting new applicants. As moving correct replicas is unnecessary, the overhead of resuming the chain is confined to placing only the failed replicas. To restore the initial degree of resiliency we must also fetch the *logs* from the correct replicas (recovery). Recovery consumes resources and may impact the performance of the other tenants. An approach to preventing this issue is to always leave a fraction of the server resources unallocated. Determining this value is a trade off between recovery time and resource efficiency. A larger value speeds up the recovery, but reduces the efficiency of resource utilization at failure-free intervals. This value must be determined based on the frequency of failures and the uptime guarantees. If failures occur when all the servers are fully allocated, it will be impossible to place the failed replicas. To avoid this problem, similar to Cheap Paxos [33], we reserve a small subset of the servers as auxiliary that are used *for placing failed replicas*.

Algorithm 1 Placement (executed by *AC*)

```
1:  while A ≠ ∅ do                                   {run until all applicants are addressed}
2:    pick applicant a ∈ A with highest dominant share   {PO_a}
3:    F_a ← S
4:    while P_a ≠ ∅ do   {run until all replicas of a are addressed}
5:      pick replica p ∈ P_a with highest dominant share   {PO_b}
6:      ElectServer (p)                                {see line 18}
7:      if replica p is not placed then
8:        ∀s ∈ E_a: update R_s                         {redeem resources}
9:        A ← A \ {a}  {Cons_2: a is rejected if any replica is not
          placed}
10:       exit loop and goto (1)
11:     else                                           {replica p is placed}
12:       P_a ← P_a \ {p}
13:   end
14:   // if here, all replicas of applicant a are placed successfully
15:   ∀s ∈ E_a : update D_s                            {PO_c}
16:   A ← A \ {a}                                      {a is admitted}
17: end

18: ElectServer (p):
19: while F_a ≠ ∅ do
20:   pick server s ∈ F_a with the lowest dominant share   {PO_c}
21:   if p ≤ R_s then   {Cons_1: a server is found to place replica}
22:     R_s = R_s − p
23:     E_a ← E_a ∪ {s}                  {update the set of elected servers}
24:     F_a ← F_a \ {s}                                {Cons_3}
25:     exit and return (replica p is placed)
26: end
27: return (replica p is not placed)                  {no server is found}
```

4.2 Work Conservation Phase

Tenants that have miss-estimated their SLAs at admission time, will need resources above or below their reservations at runtime. Filo monitors usage at runtime and temporarily re-distributes resources to address miss-estimations. Filo's objective at this phase is to frequently and fairly adjust allocations, and maximize global utilization without over-allocating. This phase executes in control intervals of a few seconds. After the resource adjustments, budgets of the token buckets must also be updated properly. Filo assigns either the *idle* or the *unallocated* resources to the demanding tenants. To understand the difference note that we only used the *unallocated* resources to admit tenants. Some tenants may not use all of their resources at runtime; these are allocated but *unused* resources. *Idle* resources include both the *unallocated* and the *unused* ones. Thus, by allocating from *idle* resources, Filo has more resources to (re-)use, but is subject to temporary SLA violations, which cannot happen by allocating only from *unallocated* resources. Accepting this risk is a decision left to Filo's operator, to which our algorithms are agnostic.

Example setup extended. We elaborate on our example from previous section to clarify this phase. Lets assume a_1's admission SLA is 100 reqs/sec that is equivalent to

the resource usage shown in Table 1. Thus for example on replica p_1 each request costs (0.01 CPU, 0.01 Gbps, 1 storage IO). At runtime, a_1 realizes that in addition to 100 requests, it needs to submit 10 more reqs/sec (110 in total). This phase tries to maximize the number of additional requests it can grant to a_1, called its *utility*, while being fair to all the other tenants that demand extra requests (note that at the end we may only be able to grant 3 requests to a_1, although it wishes for 10).

More precisely, we define $utility(t)$ to be the number of t's extra granted requests at runtime. During this phase, t's feasibility region is limited to E_t: in other words extra requests are granted if sufficient free resources on E_t exist (**Cons**$_4$). Computing new allocations quickly is crucial for reducing the intervals in which resources remain idle. In the next section we first show how this phase can be done using a centralized algorithm (§ 4.2.1). Centrally controlling and re-assigning resources is computationally intensive; moreover, its overhead increases with the size of the system (see § 6). To alleviate this overhead, we propose two new distributed algorithms (Sections 4.2.2 and 4.2.3). *Our key insight in designing distributed algorithms is*: servers do not need global visibility over *all* the other servers to dynamically adjust allocations as each tenant's replicas are placed on a small set of servers. Servers that manage the same consensus group need to coordinate with each other; hence, servers coordinate locally, and they do not need a centralized oracle. The *distributed controller* implements those algorithms.

4.2.1 Centralized DRF (C-DRF)

As a baseline, we first illustrate `C-DRF` (Alg. 2). `C-DRF` maximizes tenant utilities while equalizing their dominant shares. Differently from `DRF` [20], `C-DRF` takes the indivisibility of the demand profiles into account and is subject to **Cons**$_4$, which is not previously addressed [20, 18]. `C-DRF` has two inputs, vector $\vec{\mathbf{R}}$ and set T. To produce $\vec{\mathbf{R}}$, we concatenate the free resource vectors of the servers, $|\vec{\mathbf{R}}| = m \times k$ (in our example $\vec{\mathbf{R}} = (2,2,250,\ 2,2,400,\ 8,4,100,\ 4,2,250)$). Similarly, we extend t's demand profile as:

$$\vec{\mathbf{P_t}} = p_1\ .\ p_2\ .\ ..\ .\ p_m \tag{4}$$

Any server not in E_t is presented by $\vec{\mathbf{0}}$ in Eq 4 (assume a_1 is admitted to $E_{a_1} = \{1,3\}$ then in the granularity of one request we have $\vec{\mathbf{P_{a_1}}} = (0.01,0.01,1,\ 0,0,0,\ 0.03,0.01,1,\ 0,0,0)$): recall that the feasibility region of a tenant during this phase is restricted to the servers it is placed on.[4] We use Eq 5 to calculate t's dominant share (line 3):

$$D_t = \max_{i=1}^{k\times m} (\vec{\mathbf{P_t}} \times utility(t))_i / (\vec{\mathbf{R}})_i \tag{5}$$

In this formula $utility(t)$ is multiplied by each member of $\vec{\mathbf{P_t}}$, producing a new vector of size $m \times k$. As $utility(t)$ is updated after each allocation (line 6), D_t must be recalculated (line 7). Servers periodically send their R_s vectors to *AC*, which centrally executes `C-DRF`. T in Alg. 2 includes only the tenants that request extra resources at runtime.

As we will see in § 6.3, `C-DRF` is computationally intensive. In the next two sections we propose two efficient distributed algorithms.

Algorithm 2 C-DRF $(\vec{\mathbf{R}}, T)$

```
1:  ∀t ∈ T : utility(t) = 0
2:  while T ≠ ∅ do
3:      pick tenant t with the lowest dominant share
4:      if P_t ≤ R then
5:          R = R − P_t
6:          utility(t) = utility(t) + 1
7:          update D_t
8:      else
9:          T ← T \ t                    {E_t is saturated relative to t}
10: end
```

4.2.2 Head-DRF

`Head-DRF` (Alg. 3), our first distributed algorithm, is composed of two phases. During the allocation phase utilities are calculated by the head servers only using a local execution of `C-DRF`, and disseminated to the followers for voting. At voting phase followers cast their votes by prioritizing the allocations that maximize their local resource utilization (**PO**$_d$).[5]

Allocation Phase is executed by all the head servers in parallel. Server s is a head server if it hosts the head replica of *at least* one tenant. Server s uses Eq 6 to calculate its perspective of free resources in the cluster, $\vec{\mathbf{R}}_\mathbf{s}$:

$$\vec{\mathbf{R}}_\mathbf{s} = R_1\ .\ R_2\ .\ ..\ .\ R_m \tag{6}$$

Any server that has no tenant in common with s is presented by $\vec{\mathbf{0}}$ in Eq 6. ($|\vec{\mathbf{R}}_s| = k \times m$). Each server periodically sends its R vector to all the servers that host the head replicas of its tenants. Demand profiles are extended using Eq 4, and their dominant shares are calculated using Eq 5, where $\vec{\mathbf{R}}_s$ replaces $\vec{\mathbf{R}}$. Head servers use this data to execute `C-DRF` locally, and compute the utilities. Head servers then calculate the potential allocations for each tenant (u_t-line 4), and propose them to the relevant followers. The proposed allocation for tenant t is accepted only if *all* the servers in E_t vote for its acceptance (lines 3—6).

Given that head servers lack the global visibility over the entire cluster and may concurrently consider common servers for allocations, server resources maybe

[4] For simplicity we have not modified servers' resource vectors. Note that after admission servers have fewer resources.

[5] NTP is used to synchronize control intervals, allocation, and voting phases.

over-allocated. The *voting phase* is designed to reduce the amount of over-allocations to zero.

Voting Phase is executed by all the servers in parallel. Each server receives exactly one proposal, u_t, for each of its tenants. At the beginning of the voting phase a server has received the proposals for all the tenants for which it hosts one of their replicas. Servers follow $\mathbf{PO}_d$ to cast their votes by prioritizing the tenant with the highest dominant share (line 9). Dominant shares are calculated by Eq 5, where $\vec{\mathbf{R}}_\mathbf{s}$ replaces $\vec{\mathbf{R}}$. A server accepts a proposal only if sufficient free resources are available (line 10). Proposed allocations are unbreakable, meaning that servers can only vote for the whole allocation (u_t).

Followers are at the intersection of the disconnected perspectives of the head servers. By voting based on their local free resources, they protect themselves from over-allocations. Rejecting *full* proposals (u_t), however, might induce a low utilization of the servers. We next propose `All-DRF` to alleviate this inefficiency.

Algorithm 3 Head-DRF

1: T_{hs}: is the set of tenants that their head replica is on s.
2: *Allocation Phase: Executed by $s \in S$ if s is head for some t*
3: $\{utility(t)\} \leftarrow$ **C-DRF**$(\vec{\mathbf{R}}_s, T_{hs})$
4: $\forall t \in T_{hs}$: $u_t = (P_t \times utility(t))$,
5: $propose(u_t)$ to t's followers ,
6: $accept(u_t)$ **if** all replicas voted *accept*
7: *Voting Phase: Exe. by $\forall s \in S$*
8: **while** *true* **do**
9: **pick** u_t with the highest dominant share $\{\boldsymbol{PO}_d\}$
10: **if** $u_t \leq R_s$ **then** $\{else\ reject\}$
11: $R_s = R_s - u_t$
12: $vote(u_t, accept)$
13: **end**

4.2.3 All-DRF

`All-DRF` (Alg. 4) is also composed of two phases. Differently from `Head-DRF`:

Allocation Phase is executed by all servers in parallel (hence the name). Each server executes `C-DRF` locally and sends its utilities to the relevant head servers. Differently from `Head-DRF`, servers propose utilities ($utility_t$) rather than the full allocations (u_t). Similarly to `Head-DRF` there is a high chance of over-allocation, which is eliminated during the voting phase.

Voting Phase is executed by the head servers only. For each t, a head server chooses the minimum of the utilities proposed by all the servers in E_t. As we will see in the experiments the allocations obtained by `All-DRF` are often closer to `C-DRF`.

4.2.4 Comparison

Communication complexity. We compare complexities of our algorithms assuming that at the start of a control interval, servers have sent their R_s vectors to *AC* in `C-DRF` and to head servers in `Head-DRF`. Assuming that tenants send their runtime profiles to *AC*, one communication step is sufficient to finalize the allocations for `C-DRF`: (1) *AC* runs `C-DRF`, and broadcasts its allocations to head servers and tenants, to tune their token buckets. Assuming that tenants send their runtime profiles to head servers, 3 steps are necessary to finalize the allocations for both `Head-DRF` and `All-DRF`. In `Head-DRF`: (1) head servers run `C-DRF` and broadcast their allocations to the followers, (2) followers send their votes to the head servers, and (3) head servers summarize the votes, tune their token buckets, and inform tenants. `All-DRF` is measured similarly.

Algorithm 4 All-DRF

1: T_s: is the set of tenants that any of their replicas is on s.
2: *Allocation Phase: Executed by all $s \in S$*
3: $\{utility(t)\} \leftarrow$ **C-DRF**$(\vec{\mathbf{R}}_s, T_s)$
4: $\forall t \in T_s$: $propose(utility(t)_s)$ to t's head server
5: *Voting Phase: Exec. by $s \in S$ if s is head for some t*
6: $\forall t \in T_{hs}$: $utility(t) = \min_{\forall s \in E_t}\{utlity(t)_s\}$

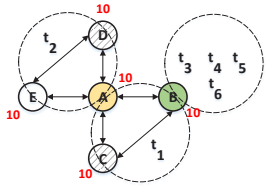


Figure 1: An example scenario for § 4.2.4

Efficiency. Distributed algorithms may diverge from the centralized approach. Consider `All-DRF` in Fig 1, with 5 servers each with 1 resource of 10 free units. t_1 as $(1,1,1,0,0)$ is placed on [A,B,C(head)], and t_2 as $(1,0,0,1,1)$ on [A,D(head),E]. At the end of the allocation phase, C,D,E assign 10, A assigns 5, and *B* assigns 2 utilities to their tenants (*B* is shared by other tenants). Head servers in parallel finalize the allocations: t_2 and t_1 receiving 5 and 2 utilities each. *A* has 3 free units that t_2 could use in an ideal situation (the values can be easily obtained by walking through the algorithm). Our evaluations will show that this inefficiency is negligible due to the large number of tenant replicas packed on a server.

4.3 Coordinating the two phases

To avoid over-allocation of the resources due to the concurrent execution of the admission and work conservation phases, *AC* notifies the elected servers before finalizing the admissions. If the servers have simultaneously allocated their resources, they either immediately terminate their allocations or ask *AC* to delay the admissions until the temporary allocations expire.

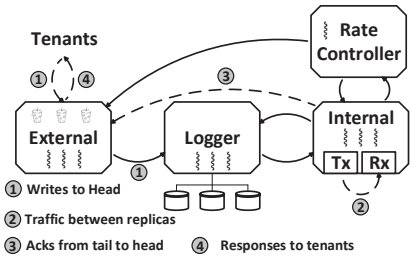


Figure 2: Internal architecture of a replica server. Dashed arrows refer to inter-server communications.

5 Implementation

Filo is written in C/C++ (11.5 KLOC). Each server is implemented as a pipeline of 3 multi-threaded stages (Fig 2). The *external* stage handles connections of tenants with the head replicas. The *Logger* stage persists data on the storage device, and the *internal* stage handles connections among servers. The number of threads, MPL (Multi Programming Level) at each stage is 1 to 4. Each server also has a thread for the distributed controller. Communications among the *stages* is based on shared memory. We optimized network performance by using jumbo frames (8968 B), by disabling Nagle algorithm, and by enabling Receive Side Scaling (RSS) [24]. In some experiments, the RSS mapping resulted in an uneven distribution, at which point despite the availability of free cores the performance was capped. We also implemented a simulator in C/C++ (2.5 KLOC) to test the resource allocator in larger configurations.

6 Evaluation

We evaluate Filo's performance in failure-free scenarios. In § 6.1 we extensively evaluate performance under various workloads, which is needed for bounding SLA guarantees, and calculating costs. Then we feed our findings into the simulator and evaluate the placement algorithm (§ 6.2), and the distributed controller (§ 6.3) with various number of applicants. In § 6.4 we demonstrate the effect of our resource allocator in the testbed.

Testbed. We ran the experiments in a cluster of 10 Dell DCS7220N servers each with two 10-core Intel(R) Xeon(R) CPU E5-2470 v2, 2.40 GHz CPUs, a 10 Gbps Mellanox ConnectX-3 Pro NIC, and 128 GB of RAM with hyper threading enabled (due to the administrative requirements disabling hyper threading was not an option.) and two HDDs (Seagate ST2000NM0033-9ZM175). Our servers use Microsoft Windows Server 2012 Data center. Unless mentioned otherwise, we ran each experiment for 60 seconds. Our graphs report the average application-level throughput in number of reqs/sec (throughput in network bandwidth can be directly calculated), total CPU utilization, and CPU utilization of the busiest logical core on one selected server.

6.1 Service benchmarking

Fig 3 shows Filo's performance in the following setup.

Setup. We vary request sizes from 64 B to 32 KB, and *MPL* from 1 to 4 for *in-mem* and *persistent* modes. In the *persistent* mode data is asynchronously transferred to the storage device and 2 (with $MPL \leq 2$) and 4 (with $MPL \geq 3$) threads are created by the *Logger*(s). Tenants send their requests in an open loop bounded by a *pending window* (*e.g.*, with a window of 10, a tenant can have at most 10 outstanding requests). We varied window size until peak performance was reached. In each setting the number of tenants is equal to *MPL*, and each has 3 replicas on 3 servers. Note that requests of a specific tenant at any given point must be handled by one thread at most for safety. One thread, however can handle the requests of many tenants. Each vertical CPU bar shows the total CPU (horizontal line) and the CPU of the busiest core.

General Results. The *in-mem* mode has higher throughput than the *persistent* mode, as the latter is capped by the capacity of the storage device and the *Logger*'s CPU. The *in-mem* mode has lower latency (1-5 msec) than the *persistent* mode (1-8 msec).

In-mem mode. As *MPL* increases, throughput increases for 64 B-4 KB requests. For 8 KB-32 KB requests, throughput increases when *MPL* (number of tenants effectively) increases from 1 to 2, and then drops: as the number of tenants increases, so does the number of outstanding requests; resources used for receiving requests are now spent for receiving requests from new tenants, exchanging new acks, and responses. As an example for *in-mem*, 2 KB, $MPL = 3$, and throughput of 150 Kreq/s, in addition to 150K requests, 150K acks, and 150K responses are transmitted among the servers, and to the tenants respectively (overhead not shown in the graphs). Often one core is saturated by the interrupts caused by sending and receiving network messages.

Persistent mode. Throughput increases as *MPL* increases. With $MPL \leq 2$ each server has only one *Logger*, through which all the threads in the storage stage transfer their requests to the disk. When *MPL* increases from 1 to 2, *group commit* improves and throughput increases (the reason our throughput surpasses IoMeter [25]). *Logger* is a point of synchronization, and eventually saturates its core for the $\leq$2 KB requests and disk's bandwidth for requests $\geq$4 KB. To scale with $MPL \geq 3$ we added a second *Logger* to each server (latency was 1-8 msec).

Aiding the admission phase. We use our numbers to bound (a) SLAs, and (b) the number of tenants with specific admission requests that can be packed on a server. To understand (a) notice that for example Filo can at most promise 76 K (64 B, *in-mem*) reqs/sec to a single tenant.

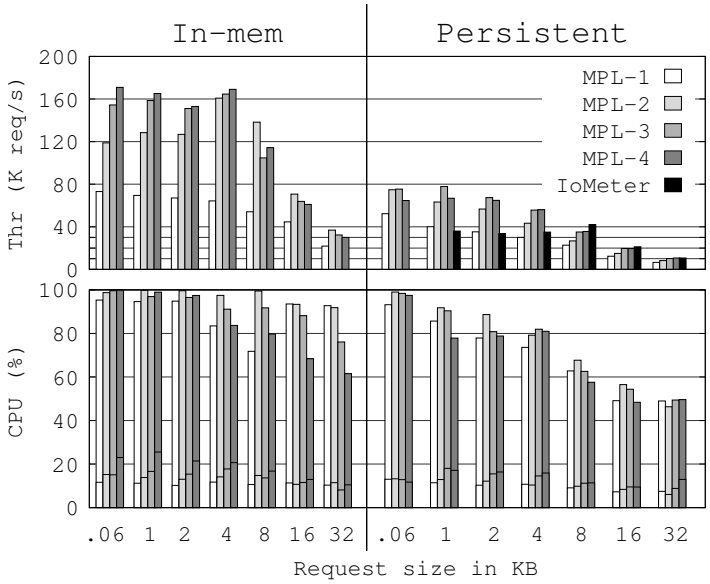


Figure 3: Evaluating Filo's performance.

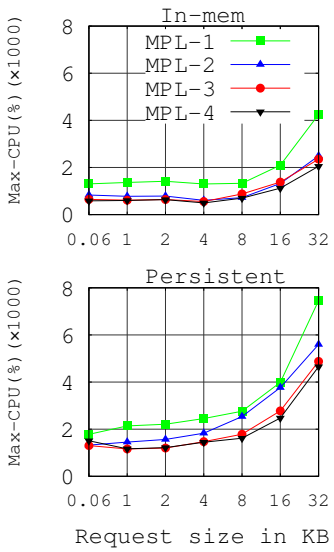


Figure 4: CPU costs, Y-axis is magnified by 1000 for readability (§ 6.1.1).

Promising SLAs above this value to one tenant is meaningless: requests of a specific tenant must be handled sequentially, therefore the peak SLA for a tenant is determined by $MPL = 1$. In other words, multiple threads in a replica can not handle the same tenant's requests. To understand (b) notice that for example a server can process 170 K (64 B, *in-mem*) reqs/sec at most. Thus the aggregate SLAs of all the tenants on one server for 64 B reqs should not exceed this value.

6.1.1 Calculating Resource Costs

We use our numbers from § 6.1 to estimate the costs of different workloads on CPU, network bandwidth, and storage IO, which are used for translating SLAs to resource costs.

CPU. We use Fig 3 to obtain the costs on the saturated CPU as shown in Fig 4. Y-axis is magnified by a factor of 1000 for readability. The cost on the total CPU can be obtained similarly. For all *MPL* the overhead on the CPU increases as the size of the requests increases. Larger requests result in higher number of interrupts and therefore higher cost on the CPU; this is because larger requests are broken into more frames (note MTU). The CPU cost in the *persistent* mode is higher than *in-mem* due to *Logger*'s overhead.

Storage IO. Throughputs in Fig 3 benefit from the *group commit* offered by *Logger*. We must exclude this benefit when the system is not highly loaded. Thus to calculate storage costs we use IoMeter's values instead. At its maximum, IoMeter measures ∼40 K IOs for 1-8 KB requests. With a peak value of 40 K IOs, requests ≤ 8 *KB* translate to one IO each. 16 KB and 32 KB requests translate to 2 and 4 IOs respectively. To understand note that for 16 KB and 32 KB requests, IoMeter measures 20 K and 10 K IOs respectively. Given that *Logger* writes each request twice (spread across two storage devices), we multiply the number of IOs by 2 .

Network bandwidth. To calculate network costs we use request sizes directly (e.g., a 1 KB request costs 1 KB network bandwidth).

Inaccuracies in the cost model. The assessment space of a complex system such as Filo is extremely large, and Fig 3 covers only a tiny fraction of this space. Hence, calculating costs using only Fig 3 is prone to inaccuracies that can lead to SLA violations. To prevent SLA violations we can base our admission decisions on x % of the peak throughput. For example if Filo can provide 76 K (64 B, *in-mem*) reqs/sec, with x as 80% a tenant can at most be promised $\approx$ 60 K reqs/sec. If x is chosen conservatively, resources will be under-allocated, but work conservation phase will compensate for it.

Generalizing costs. For workloads not covered by our evaluations costs are obtained by scaling.

Recalculating costs. As the hardware specifications or the implementation changes, performance must be reassessed, and cost estimations must be recalculated.

6.2 Admission Phase

We use our simulator to evaluate the placement algorithm (§ 4.1) in on-line and off-line settings; applicants arrive one at a time in the former and all at once in the latter. We order tenants with increasing and decreasing dominant shares in on-line and off-line settings respectively, to account for the worst and best cases for each.

Setup. We assume 10 servers and up to 150 applicants with a mixture of requirements. We use uniform distributions to choose durability mode (*in-mem* or *persistent*), requests sizes (64 B to 32 KB), and replication degree

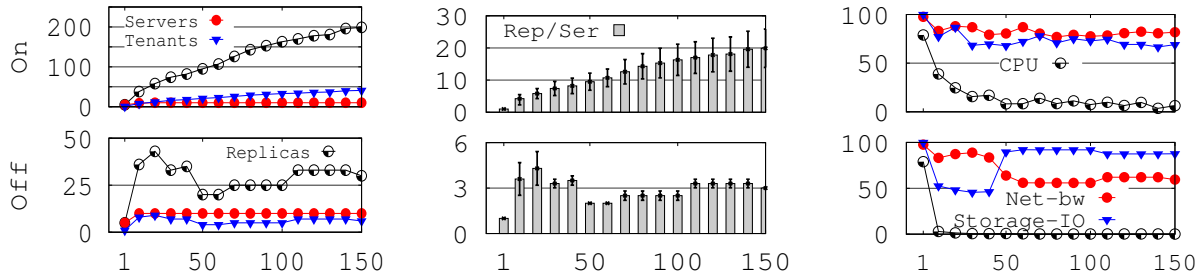


Figure 5: Placement algorithm in on-line and off-line settings. X-axis shows the number of applicants (§ 6.2).

(3,5,7). To simulate our testbed characteristics the SLA for each applicant is bound by our evaluations in § 6.1 (*e.g.*, with 64 B, *in-mem* the peak SLA an applicant can be promised is 76 K reqs/sec). Replica placement on each server is bound by our numbers from Fig 3. We use our cost model from § 6.1.1 to translate admission requests to resource profiles.

Results. Fig 5 shows results in 3 graphs. Left: number of used servers, admitted tenants, and replicas; Middle: number of replicas per server (average, 95% confidence interval); Right: percentage of free resources at the end of the admission phase. In both on-line and off-line modes the applicants are denied admission if any of the resources (in this case, CPU) is saturated. In the on-line mode as the number of applicants increases, fewer tenants are admitted but the number of replicas increases (e.g., with 150 applicants, $\approx$ 200 replicas are packed on 10 servers). On-line case admits more tenants and packs more replicas. As expected, off-line mode selects fewer applicants but the ones that maximize resource utilizations. Although off-line results in more efficient allocations (Fig 5(right)), it is hard to know the list of applicants a priori. In both cases our placement algorithm is efficient and all the 10 servers are efficiently utilized.

6.3 Work Conservation Phase

We use our simulator to compare `C-DRF`, `Head-DRF` and `All-DRF` for computation and communication overheads, utilization, fairness, and SLA violations in the on-line setup of § 6.2. We consider a case where half of the tenants are not using their reservations, and the other half demand extra rates. Resource profiles for the work conservation phase are equal to the admission profiles, but in the granularity of one request (see § 4.2 for explanation).

Computation overhead. Fig 6(a) shows the time it takes to compute allocations. As the number of tenants increases, compared to the centralized, distributed algorithms perform faster ($\sim$ 5 times at best). We argued in § 4.2 that computing allocations quickly is important for reducing the periods in which resources remain idle.

Communication overhead. Fig 6(b) shows the number of messages. Compared with `C-DRF`, distributed algorithms exchange about 8 times more messages (150 msgs at worst). Given the amount of service-level messages, this overhead is negligible. These messages can further be piggybacked on the service messages.

Resource utilization. Fig 6(c) shows the aggregate amount of the free resources before and after work conservation (we have eliminated network and storage due to space limits). Our distributed algorithms result in allocations that compare closely with the `C-DRF`: `All-DRF` is 95% as efficient as `C-DRF`, and `Head-DRF` about 75%.

Fairness. Fig 6(d) shows fairness. As a fairness criterion, we have used normalized standard deviation for the *utility*, with `C-DRF` as the reference. An algorithm is more fair if this value is smaller. For example if with a total budget of 15 reqs/sec and 3 tenants, an algorithm allocates 5 to each tenant, it is more fair than an algorithm that allocates 10, 5, and 0. Unlike `Head-DRF`, `All-DRF`'s fairness compares closely to that of the `C-DRF`. This is because proposals can only be accepted as a whole in `Head-DRF`, but partially in `All-DRF`.

SLA violations. To absorb the exceeding demands of the tenants we used all the *idle* resources. `C-DRF` and `All-DRF` are subject to above 95% SLA violations. This is because these two algorithms are very efficient at allocating resources. Whether to use reserved resources is part of system's policy and does not affect the semantics of our algorithms (see § 4.2 for details).

To conclude, `All-DRF` compares well with `C-DRF` in efficiency and fairness and in addition is about 5 $\times$ faster. `Head-DRF` has lower computation time, but it is 70% as efficient as `C-DRF` and is less fair. We observed similar results in other settings, where for example replica demand vectors were skewed.

6.4 Performance Isolation

Fig 7 shows the impact of the resource allocator and rate limiters with 3 tenants, 3 replicas each in the *persistent* mode, and SLA of 6.5 K-1 KB reqs/sec each. Assume *A* and *B* will need more resources at runtime. In the first 4 minutes only *B* and *C* are in the system, and in the first 2 minutes rate limiters are disabled. At min 1, *B* increases its rate above its SLA and affects *C*'s performance. At min 2 rate limiters are enabled and hence SLAs restored. At min 3, *C* voluntarily reduces its rate for the next 3 minutes. At min 4 we activate `All-DRF`, admit *A*, but intentionally leave its runtime extra demand

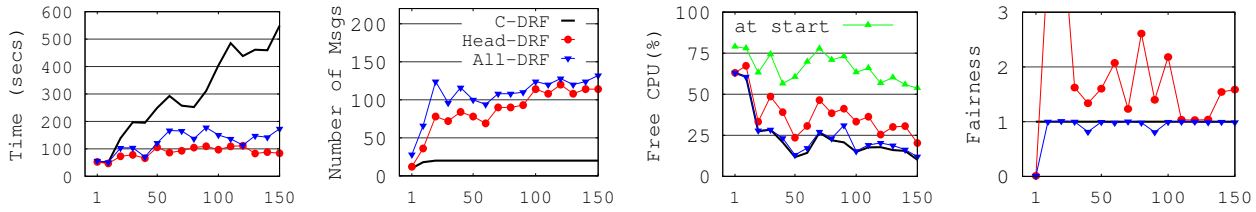


Figure 6: Evaluating `C-DRF`, `Head-DRF`, `All-DRF`. X-axis shows the number of applicants (§ 6.3 for details).

out of `All-DRF`'s sight. The algorithm grants to *B* all of *C*'s underutilized resources neglecting *A*. At min 5, *A* is considered by `All-DRF` as well. At this point the underutilized resources of *C* are fairly divided between *A* and *B*. At min 6, *C* restores its SLA rate.

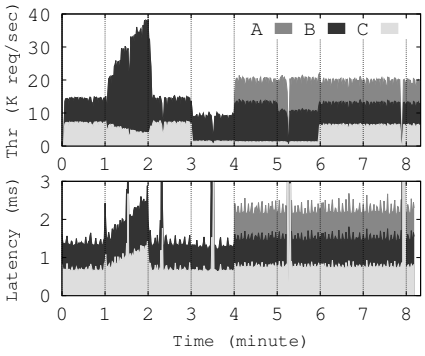


Figure 7: The impact of the distributed controller.

7 Related work

Consensus. Consensus is widely studied in the database and distributed systems communities [7, 30, 17, 16, 14, 19, 34]. Although some works have studied the colocation of consensus instances on shared servers [12, 37, 5], no one has considered the problem of efficient replica placement and the impact of sharing on the performance of individual instances. Filo is the first system to provide consensus as a service for the multi-tenant environment of the cloud platforms, while providing SLA guarantees, performance isolation, and efficiency of resource utilization. Similarly to Filo, [36, 4] use Chain Replication [42] for providing ordering and storage guarantees.

Token Buckets. Filo uses token buckets for rate limiting. Token buckets are previously used in network rate limiting [48, 1]. Filo's approach in determining and constantly updating the budget of the token buckets is novel, which is realized by its resource allocator component.

Distributed Controller. Filo is similar to Retro [35] in its objectives for ensuring performance isolation and efficient resource utilization. Unlike Retro, Filo uses a distributed controller for dynamically tuning the rate limiters. Few works have considered distributed rate tuning. [41] proposes a distributed approach, where each node is equipped with a rate limiter. A user is given an aggregate global budget, and each rate limiter's bucket is initialized with this budget. Each node removes tokens based on its own usage rate and the estimated sum of the usage rates at all the other nodes. This work is later extended in [47]. Consensus groups in Filo are composed of small number of servers that can locally coordinate their resource usage and achieve high resource efficiency without needing visibility over the entire cluster. Moreover, given the distributed model of the consensus and the connection links that are already established, designing a distributed controller in Filo had no additional cost. Our evaluations showed that our distributed algorithms exchange a small amount of messages to coordinate and finalize the allocations. Compared with centralized controllers, distributed controllers are computationally less intensive and faster.

Resource Allocation. Multi-resource allocation is a multi-resource bin packing problem [6, 15, 22, 27, 29, 45]. Filo's allocation algorithms are inspired by DRF [20, 18, 40]. DRF is not guaranteed to converge to the global optimum; [39, 21] propose heuristics algorithms for converging to the global optimum, which are computationally intensive and not suitable in our context.

Empirical quantification. We used an empirical approach to quantifying Filo prior to its launch [50, 26, 1]. Our strategy can be enhanced further by dynamically modifying the assessments at runtime.

8 Conclusion

We presented Filo, the first system to provide multi-tenant consolidated consensus as a cloud service. We argued that providing performance guarantees and isolation is particularly important in the cloud platforms where tenants share and compete over resources. We proposed a novel placement algorithm for admitting tenants, and two distributed algorithms for efficient and fair allocation of free resources at runtime. Our algorithms exploit the nature of the consensus service, and while being much faster provide comparable efficiency and fairness compared with the centralized algorithms.

9 Acknowledgments

We thank Miguel Castro, Austin Donnelly, Greg O'Shea, Richard Black, reviewers, and Mohit Aron for their feedback and support.

References

[1] S. ANGEL, H. BALLANI, T. KARAGIANNIS, G. OSHEA, AND E. THERESKA End-to-end performance isolation through virtual datacenters. In: *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association. 2014.

[2] *Apache BookKeeper*. Apache. 2014. URL: `http://bookkeeper.apache.org/`.

[3] *Apache HBase*. Apache. 2015. URL: `http://hbase.apache.org/`.

[4] M. BALAKRISHNAN, D. MALKHI, T. WOBBER, M. WU, V. PRABHAKARAN, M. WEI, J. D. DAVIS, S. RAO, T. ZOU, AND A. ZUCK Tango: Distributed data structures over a shared log. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013.

[5] S. BENZ, P. J. MARANDI, F. PEDONE, AND B. GARBINATO Building global and scalable systems with atomic multicast. In: *Proceedings of the 15th International Middleware Conference*. ACM. 2014.

[6] A. A. BHATTACHARYA, D. CULLER, E. FRIEDMAN, A. GHODSI, S. SHENKER, AND I. STOICA Hierarchical scheduling for diverse datacenter workloads. In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013.

[7] K. P. BIRMAN, R. VAN RENESSE, ET AL. *Reliable distributed computing with the Isis toolkit*. Vol. 85. IEEE Computer society press Los Alamitos, 1994.

[8] W. J. BOLOSKY, D. BRADSHAW, R. B. HAAGENS, N. P. KUSTERS, AND P. LI Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In: *NSDI*. 2011.

[9] M. BURROWS The Chubby lock service for loosely-coupled distributed systems. In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006.

[10] F. CHANG, J. DEAN, S. GHEMAWAT, W. C. HSIEH, D. A. WALLACH, M. BURROWS, T. CHANDRA, A. FIKES, AND R. E. GRUBER Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008).

[11] B. CHARRON-BOST, F. PEDONE, AND A. SCHIPER *Replication: theory and Practice*. Vol. 5959. springer, 2010.

[12] B. DARNELL *Scaling Raft*. 2015. URL: `http://www.cockroachlabs.com/blog/scaling-raft/`.

[13] G. DECANDIA, D. HASTORUN, M. JAMPANI, G. KAKULAPATI, A. LAKSHMAN, A. PILCHIN, S. SIVASUBRAMANIAN, P. VOSSHALL, AND W. VOGELS Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007).

[14] D DOLEV, AND H. STRONG *Distributed commit with bounded waiting*. IBM Thomas J. Watson Research Division, 1982.

[15] D. DOLEV, D. G. FEITELSON, J. Y. HALPERN, R. KUPFERMAN, AND N. LINIAL No justified complaints: On fair sharing of multiple resources. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. ACM. 2012.

[16] C. DWORK, N. LYNCH, AND L. STOCKMEYER Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988).

[17] M. J. FISCHER, N. A. LYNCH, AND M. S. PATERSON Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985).

[18] E. FRIEDMAN, A. GHODSI, AND C.-A. PSOMAS Strategyproof allocation of discrete jobs on multiple machines. In: *Proceedings of the fifteenth ACM conference on Economics and computation*. ACM. 2014.

[19] H. GARCIA-MOLINA Elections in a distributed computing system. *Computers, IEEE Transactions on* 100, 1 (1982).

[20] A. GHODSI, M. ZAHARIA, B. HINDMAN, A. KONWINSKI, S. SHENKER, AND I. STOICA Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In: *NSDI*. Vol. 11. 2011.

[21] R. GRANDL, G. ANANTHANARAYANAN, S. KANDULA, S. RAO, AND A. AKELLA Multi-resource packing for cluster schedulers. In: *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM. 2014.

[22] A. GUTMAN, AND N. NISAN Fair allocation without trade. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*. International Foundation for Autonomous Agents AND Multiagent Systems. 2012.

[23] P. HUNT, M. KONAR, F. P. JUNQUEIRA, AND B. REED ZooKeeper: Wait-free Coordination for Internet-scale Systems. In: *USENIX Annual Technical Conference*. Vol. 8. 2010.

[24] *Introduction to Receive Side Scaling*. Microsoft. URL: `https://msdn.microsoft.com/en-us/library/windows/hardware/ff556942(v=vs.85).aspx`.

[25] *Iometer benchmark*. Intel Corporation. 2013. URL: `http://www.iometer.org/`.

[26] V. JALAPARTI, H. BALLANI, P. COSTA, T. KARAGIANNIS, AND A. ROWSTRON Bazaar: Enabling predictable performance in datacenters. *Microsoft Res., Cambridge, UK, Tech. Rep. MSR-TR-2012-38* (2012).

[27] C. JOE-WONG, S. SEN, T. LAN, AND M. CHIANG Multiresource Allocation: Fairness–Efficiency Tradeoffs in a Unifying Framework. *Networking, IEEE/ACM Transactions on* 21, 6 (2013).

[28] M. KAPRITSOS, Y. WANG, V. QUEMA, A. CLEMENT, L. ALVISI, M. DAHLIN, ET AL. All about Eve: Execute-Verify Replication for Multi-Core Servers. In: *OSDI*. Vol. 12. 2012.

[29] I. KASH, A. D. PROCACCIA, AND N. SHAH No agent left behind: Dynamic fair division of multiple resources. *Journal of Artificial Intelligence Research* (2014).

[30] L. LAMPORT Paxos made simple. *ACM Sigact News* 32, 4 (2001).

[31] L. LAMPORT, D. MALKHI, AND L. ZHOU Reconfiguring a state machine. *ACM SIGACT News* 41, 1 (2010).

[32] L. LAMPORT, D. MALKHI, AND L. ZHOU Vertical paxos and primary-backup replication. In: *ACM symposium on Principles of distributed computing*. ACM. 2009.

[33] L. LAMPORT, AND M. MASSA Cheap paxos. In: *Dependable Systems and Networks, 2004 International Conference on*. IEEE. 2004.

[34] B. W. LAMPSON Replicated commit. In: *Circulated at a workshop on Fundamental Principles of Distributed Computing, Pala Mesa, CA*. 1980.

[35] J. MACE, P. BODIK, R. FONSECA, AND M. MUSUVATHI Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. USENIX Association, 2015.

[36] D. MALKHI, M. BALAKRISHNAN, J. D. DAVIS, V. PRABHAKARAN, AND T. WOBBER From paxos to CORFU: a flash-speed shared log. *ACM SIGOPS Operating Systems Review* 46, 1 (2012).

[37] P. J. MARANDI, M. PRIMI, AND F. PEDONE Multi-ring paxos. In: *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*. IEEE. 2012.

[38] L. E. MOSER, Y. AMIR, P. M. MELLIAR-SMITH, AND D. A. AGARWAL Extended virtual synchrony. In: *Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on*. IEEE. 1994.

[39] R. PANIGRAHY, K. TALWAR, L. UYEDA, AND U. WIEDER Heuristics for vector bin packing. *research. microsoft. com* (2011).

[40] D. C. PARKES, A. D. PROCACCIA, AND N. SHAH Beyond dominant resource fairness: extensions, limitations, and indivisibilities. *ACM Transactions on Economics and Computation* 3, 1 (2015).

[41] B. RAGHAVAN, K. VISHWANATH, S. RAMABHADRAN, K. YOCUM, AND A. C. SNOEREN Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review* 37, 4 (2007).

[42] R. van RENESSE, AND F. B. SCHNEIDER Chain Replication for Supporting High Throughput and Availability. In: *OSDI*. Vol. 4. 2004.

[43] N. SCHIPER, AND S. TOUEG A robust and lightweight stable leader election service for dynamic systems. In: *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*. IEEE. 2008.

[44] F. B. SCHNEIDER Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990).

[45] D SHAH, AND D WISCHIK Principles of resource allocation in networks. In: *Proceedings of the ACM SIGCOMM Education Workshop*. 2011.

[46] E. G. SIRER, AND D. ALTINBUKEN *Commodifying Replicated State Machines with OpenReplica*. Technical Report 1813-29009. Cornell University, 2012.

[47] R. STANOJEVI, AND R. SHORTEN Fully decentralized emulation of best-effort and processor sharing queues. *ACM SIGMETRICS Performance Evaluation Review* 36, 1 (2008).

[48] A. S. TANENBAUM Computer networks, 4-th edition. *ed: Prentice Hall* (2003).

[49] E. THERESKA, H. BALLANI, G. O'SHEA, T. KARAGIANNIS, A. ROWSTRON, T. TALPEY, R. BLACK, AND T. ZHU Ioflow: A software-defined storage architecture. In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013.

[50] B. URGAONKAR, P. SHENOY, AND T. ROSCOE Resource overbooking and application profiling in shared hosting platforms. *ACM SIGOPS Operating Systems Review* 36, SI (2002).

[51] J. YIN, J.-P. MARTIN, A. VENKATARAMANI, L. ALVISI, AND M. DAHLIN Separating agreement from execution for byzantine fault tolerant services. In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003.

Modular Composition of Coordination Services

Kfir Lev-Ari[1], Edward Bortnikov[2], Idit Keidar[1,2], and Alexander Shraer[3]

[1]Viterbi Department of Electrical Engineering, Technion, Haifa, Israel
[2]Yahoo Research, Haifa, Israel
[3]Google, Mountain View, CA, USA

Abstract

Coordination services like ZooKeeper, etcd, Doozer, and Consul are increasingly used by distributed applications for consistent, reliable, and high-speed coordination. When applications execute in multiple geographic regions, coordination service deployments trade-off between performance, (achieved by using independent services in separate regions), and consistency.

We present a system design for modular composition of services that addresses this trade-off. We implement ZooNet, a prototype of this concept over ZooKeeper. ZooNet allows users to compose multiple instances of the service in a consistent fashion, facilitating applications that execute in multiple regions. In ZooNet, clients that access only local data suffer no performance penalty compared to working with a standard single ZooKeeper. Clients that use remote and local ZooKeepers show up to 7x performance improvement compared to consistent solutions available today.

1 Introduction

Many applications nowadays rely on coordination services such as ZooKeeper [28], etcd [9], Chubby [24], Doozer [8], and Consul [5]. A coordination service facilitates maintaining shared state in a consistent and fault-tolerant manner. Such services are commonly used for inter-process coordination (e.g., global locks and leader election), service discovery, configuration and metadata storage, and more.

When applications span multiple data centers, one is faced with a choice between sacrificing performance, as occurs in a cross data center deployment, and forgoing consistency by running coordination services independently in the different data centers. For many applications, the need for consistency outweighs its cost. For example, Akamai [40] and Facebook [41] use strongly-consistent globally distributed coordination services (Facebook's Zeus is an enhanced version of ZooKeeper) for storing configuration files; dependencies among configuration files mandate that multiple users reading such files get consistent versions in order for the system to operate properly. Other examples include global service discovery [4], storage of access-control lists [1] and more.

In this work we leverage the observation that, nevertheless, such workloads tend to be highly partitionable. For example, configuration files of user or email accounts for users in Asia will rarely be accessed outside Asia. Yet currently, systems that wish to ensure consistency in the rare cases of remote access, (like [40, 41]), globally serialize all updates, requiring multiple cross data center messages.

To understand the challenge in providing consistency with less coordination, consider the architecture and semantics of an individual coordination service. Each coordination service is typically replicated for high-availability, and clients submit requests to one of the replicas. Usually, update requests are serialized via a quorum-based protocol such as Paxos [32], Zab [29] or Raft [37]. Reads are served locally by any of the replicas and hence can be somewhat stale but nevertheless represent a valid snapshot. This design entails the typical semantics of coordination services [5, 9, 28] – atomic (linearizable [27]) updates and sequentially-consistent [31] reads. Although such weaker read semantics enable fast local reads, this property makes coordination services non-composable: correct coordination services may fail to provide consistency when combined. In other words, a workload accessing *multiple* consistent coordination services may not be consistent, as we illustrate in Section 2. This shifts the burden of providing consistency back to the application, beating the purpose of using coordination services in the first place.

In Section 3 we present a system design for modular composition of coordination services, which addresses this challenge. We propose deploying a single coor-

dination service instance in each data center, which is shared among many applications. Each application partitions its data among one or more coordination service instances to maximize operation locality. Distinct coordination service instances, either within a data center or geo-distributed, are then composed in a manner that guarantees global consistency. Consistency is achieved on the client side by judiciously adding synchronization requests. The overhead incurred by a client due to such requests depends on the frequency with which that client issues read requests to *different* coordination services. In particular, clients that use a single coordination service do not pay any price.

In Section 4 we present ZooNet, a prototype implementation of our modular composition for ZooKeeper. ZooNet implements a client-side library that enables composing multiple ZooKeeper ensembles, (i.e., service instances), in a consistent fashion, facilitating data sharing across geographical regions. Each application using the library may compose ZooKeeper ensembles according to its own requirements, independently of other applications. Even though our algorithm requires only client-side changes, we tackle an additional issue, specific to ZooKeeper – we modify ZooKeeper to provide better isolation among clients. While not strictly essential for composition, this boosts performance of both stand-alone and composed ZooKeeper ensembles by up to 10x. This modification has been contributed back to ZooKeeper [21] and is planned to be released in ZooKeeper 3.6.

In Section 5 we evaluate ZooNet. Our experiments show that under high load and high spatial or temporal locality, ZooNet achieves the same performance as an inconsistent deployment of independent ZooKeepers (modified for better isolation). This means that our support for consistency comes at a low performance overhead. In addition, ZooNet shows up to 7.5x performance improvement compared to a consistent ZooKeeper deployment (the "recommended" way to deploy ZooKeeper across data centers [13]).

We discuss related work in Section 6, and conclude the paper, and discuss future directions in Section 7.

In summary, this paper makes the following contributions:

- A system design for composition of coordination services that maintains their semantics.
- A significant improvement to ZooKeeper's server-side isolation and concurrency.
- ZooNet – a client-side library to compose multiple ZooKeepers.

2 Background

We discuss the service and semantics offered by coordination services in Section 2.1, and then proceed to discuss possible ways to deploy them in a geo-distributed setting in Section 2.2.

2.1 Coordination Services

Coordination services are used for maintaining shared state in a consistent and fault-tolerant manner. Fault tolerance is achieved using replication, which is usually done by running a quorum-based state-machine replication protocol such as Paxos [32] or its variants [29, 37].

In Paxos, the history of state updates is managed by a set of servers called *acceptors*, s.t. every update is voted on by a quorum (majority) of acceptors. One acceptor serves as *leader* and manages the voting process. In addition to acceptors, Paxos has *learners* (called *observers* in ZooKeeper and *proxies* in Consul), which are lightweight services that do not participate in voting and get notified of updates after the quorum accepts them. In the context of this paper, acceptors are also (voting) learners, i.e., they learn the outcomes of votes.

Coordination services are typically built on top of an underlying key-value store and offer read and update (read-modify-write) operations. The updates are linearizable, i.e., all acceptors and learners see the same sequence of updates and this order conforms to the real-time order of the updates. The read operations are sequentially consistent, which is a weaker notion similar to linearizability in that an equivalent sequential execution must exist, but it must only preserve the program order of each individual client and not the global real-time order. A client can thus read a stale value that has already been overwritten by another client. These weaker semantics are chosen in order to allow a single learner or acceptor to serve reads locally. This motivates using learners in remote data centers – they offer fast local reads without paying the cost of cross data center voting.

As an aside, we note that some coordination service implementations offer their clients an asynchronous API. This is a client-side abstraction that improves performance by masking network delays. At the server-side, each client's requests are handled sequentially, and so the interaction is well-formed, corresponding to the standard correctness definitions of linearizability and sequential consistency.

Unfortunately, these semantics of linearizable updates and sequentially consistent reads are not composable, i.e., a composition of such services does not satisfy the same semantics. This means that the clients cannot predict the composed system's behavior. As an example, consider two clients that perform operations concurrently

as we depict in Figure 1. Client 1 updates object x managed by coordination service s_1, and then reads an old version of object y, which is managed by service s_2. Client 2 updates y and then reads an old version of x. While the semantics are preserved at both s_1 and s_2 (recall that reads don't have to return the latest value), the resulting execution violates the service semantics since there is no equivalent sequential execution: the update of y by client 2 must be serialized after the read of y by client 1 (otherwise the read should have returned 3 and not 0), but then the read of x by client 2 appears after the update of x by client 1 and therefore should have returned 5.

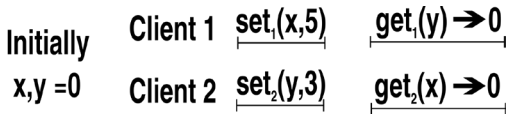


Figure 1: Inconsistent composition of two coordination services holding objects x and y: each object is consistent by itself, but there is no equivalent sequential execution.

2.2 Cross Data Center Deployment

When coordination is required across multiple data centers over WAN, system architects currently have three main deployment alternatives. In this section we discuss these alternatives with respect to their performance, consistency, and availability in case of partitions. A summary of our comparison is given in Table 1.

Alternative 1 – Single Coordination Service A coordination service can be deployed over multiple geographical regions by placing its acceptors in different locations (as done, e.g., in Facebook's Zeus [41] or Akamai's ACMS [40]), as we depict in Figure 2a. Using a single coordination service for all operations guarantees consistency.

This setting achieves the best availability since no single failure of a data center takes down all acceptors. But in order to provide availability following a loss or disconnection of any single data center, more than two locations are needed, which is not common.

With this approach, voting on each update is done across WAN, which hampers latency and wastes WAN bandwidth, (usually an expensive and contended resource). In addition, performance is sensitive to placement of the leader and acceptors, which is frequently far from optimal [39]. On the other hand, reads can be served locally in each partition.

Alternative 2 – Learners A second option is to deploy all of the acceptors in one data center and learners in others, as we depict in Figure 2b. In fact, this architecture was one of the main motivations for offering learners (observers) in ZooKeeper [13]. As opposed to acceptors, a learner does not participate in the voting process and it only receives the updates from the leader once they are committed. Thus, cross data center consistency is preserved without running costly voting over WAN. Often, alternatives 1 and 2 are combined, such as in Spanner [25], Megastore [22] and Zeus [41].

The update throughput in this deployment is limited by the throughput of one coordination service, and the update latency in remote data centers is greatly affected by the distance between the learners and the leader. In addition, in this approach we have a single point of failure, i.e., if the acceptors' data center fails or a network partition occurs, remote learners are only able to serve read requests.

Alternative 3 – Multiple Coordination Services In the third approach data is partitioned among several independent coordination services, usually one per data center or region, each potentially accompanied by learners in remote locations, as depicted in Figure 2c. In this case, each coordination service processes only updates for its own data partition and if applications in different regions need to access unrelated items they can do so independently and in parallel, which leads to high throughput. Moreover, if one cluster fails all other locations are unaffected. Due to these benefits, multiple production systems [4, 11, 18] follow this general pattern. The disadvantage of this design is that it does not guarantee the coordination service's consistency semantics, as explained in Section 2.1.

3 Design for Composition

In Section 3.1 we describe our design approach and our client-side algorithm for modular composition of coordination services while maintaining consistency. In Section 3.2 we discuss the properties of our design, namely correctness (a formal proof is given in an online Technical Report [33]), performance, and availability.

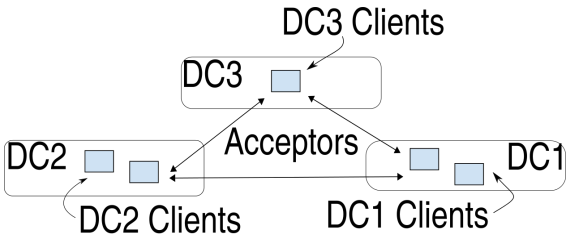
3.1 Modular Composition of Services

Our design is based on multiple coordination services (as depicted in Figure 2c), to which we add client-side logic that enforces consistency.

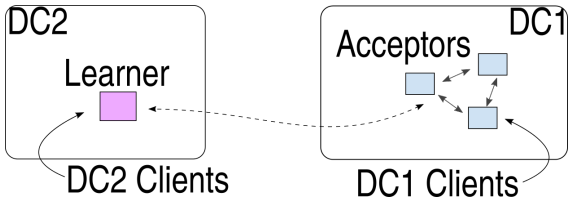
Our solution achieves consistency by injecting *sync* requests, which are non-mutating update operations. If the coordination service itself does not natively support such operations, they can be implemented using an update request addressed to a dummy object.

Alternative	Performance		Correctness	Availability during partitions	
	Updates	Reads		Updates	Reads
`Single Service`	Very slow	Fast	Yes	In majority	Everywhere
`Learners`	Slow	Fast	Yes	In acceptors	Everywhere
`Multiple Services`	Fast	Fast	No	Local	Everywhere
`Modular Composition`	Fast	Fast	Yes	Local	Local

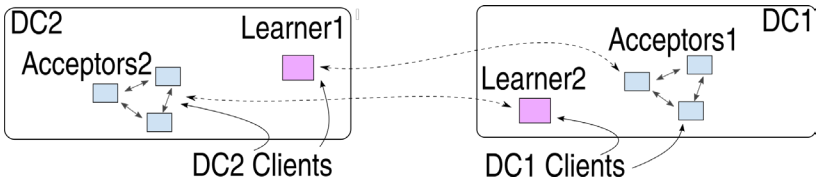
Table 1: Comparison of different alternatives for coordination service deployments across data centers. The first three alternatives are depicted in Figure 2. Our design alternative, *modular composition*, is detailed in Section 3.



(a) Single Service – Coordination service acceptors are deployed in all data centers, no single point of failure.



(b) Learners – Coordination service acceptors are deployed in one data center and learners in all other data centers.



(c) Multiple Services – A single coordination service is deployed in each data center and a learner is deployed in every other data center.

Figure 2: Different alternatives for coordination service deployment across data centers.

The client-side logic is implemented as a layer in the coordination service client library, which receives the sequential stream of client requests before they are sent to the coordination service. It is a state machine that selectively injects sync requests prior to some of the reads. Intuitively, this is done to bound the staleness of ensuing reads. In Algorithm 1, we give a pseudo-code for this layer at a client accessing multiple coordination services, each of which has a unique identifier.

An injected sync and ensuing read may be composed into a single operation, which we call *synced read*. A synced read can be implemented by buffering the local read request, sending a sync (or non-mutating update) to the server, and serving the read immediately upon receipt of a commit for the sync request. Some coordination services natively support such synced reads, e.g., Consul calls them consistent reads [6]. If all reads are synced the execution is linearizable. Our algorithm only makes some of the reads synced to achieve coordination service's semantics with minimal synchronization overhead.

Since each coordination service orders requests independently, concurrent processing of a client's updates at two coordination services may inverse their order. To avoid such re-ordering (as required, e.g., by ZooKeeper's FIFO program order guarantee), we refrain from asynchronously issuing updates to a new coordination service before responses to earlier requests arrive. Rather, we buffer requests whenever we identify a new coordination service target for as long as there are pending requests to other coordination services. This approach also guarantees that coordination service failures do not introduce gaps in the execution sequence of asynchronous requests.

3.2 Modular Composition Properties

We now discuss the properties of our modular composition design.

Algorithm 1 Modular composition, client-side logic.

```
1:  lastService ← nil            // Last service this client accessed
2:  numOutstanding ←0 // #outstanding requests to lastService
3:  onUpdate(targetService, req)
4:    if targetService ≠ lastService then
5:      // Wait until all requests to previous service complete
6:      wait until numOutstanding = 0
7:      lastService ← targetService
8:    numOutstanding++
9:    send req to targetService
10: onRead(targetService, req)
11:   if targetService ≠ lastService then
12:     // Wait until all requests to previous service complete
13:     wait until numOutstanding = 0
14:     lastService ← targetService
15:     numOutstanding++
16:     // Send sync before read
17:     send sync to targetService
18:   numOutstanding++
19:   send req to targetService
20: onResponse(req)
21:   numOutstanding−−
```

3.2.1 Correctness

The main problem in composing coordination services is that reads might read "from the past", causing clients to see updates of different coordination services in a different order, as depicted in Figure 1. Our algorithm adds sync operations in order to make ensuing reads "read from the present", i.e., read at least from the sync point. We do this every time a client's read request accesses a different coordination service than the previous request. Subsequent reads from the same coordination service are naturally ordered after the first, and so no additional syncs are needed.

In Figure 3 we depict the same operations as in Figure 1 with sync operations added according to our algorithm. As before, client 1 updates object x residing in service s_1 and then reads y from service s_2. Right before the read, the algorithm interjects a sync to s_2. Similarly, client 2 updates y on s_2, followed by a sync and a read from s_1. Since s_2 guarantees update linearizability and client 1's sync starts after client 2's update of y completes, reads made by client 1 after the sync will retrieve the new state, in this case 3. Client 2's sync, on the other hand, is concurrent with client 1's update of x, and therefore may be ordered either before or after the update. In this case, we know that it is ordered before the update, since client 2's read returns 0. In other words, there exists an equivalent sequential execution that consists of client 2's requests followed by client 1's requests, and this execution preserves linearizability of updates (and syncs) and sequential consistency of read requests, as required by the coordination service's semantics. See [33] for a formal discussion.

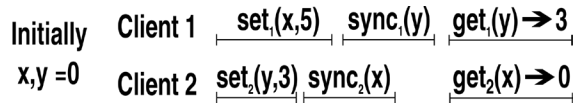


Figure 3: Consistent modular composition of two coordination services holding objects x and y (as in Figure 1): adding syncs prior to reads on new coordination services ensures that there is an equivalent sequential execution.

3.2.2 Performance

By running multiple independent coordination services, the modular composition can potentially process requests at a rate as high as the sum of the individual throughputs. However, sync requests take up part of this bandwidth, so the net throughput gain depends on the frequency with which syncs are sent.

The number of syncs corresponds to the temporal locality of the workload, since sync is added only when the accessed coordination service changes.

Read latency is low (accessing a local acceptor or learner) when the read does not necessitate a sync, and is otherwise equal to the latency of an update.

3.2.3 Availability

Following failures or partitions, each local coordination service (where a quorum of acceptors remains available and connected) can readily process update and read requests submitted by local clients. However, this may not be the case for remote client requests: If a learner in data center A loses connectivity with its coordination service in data center B, sync requests submitted to the learner by clients in A will fail and these clients will be unable to access the coordination service.

Some coordination services support state that corresponds to active client sessions, e.g., an ephemeral node in ZooKeeper is automatically deleted once its creator's session terminates. Currently, we do not support composition semantics for such session-based state: clients initiate a separate session with each service instance they use, and if their session with one ZooKeeper ensemble expires (e.g., due to a network partition) they may still access data from other ZooKeepers. Later, if the session is re-instated they may fail to see their previous session-based state, violating consistency. A possible extension addressing this problem could be to maintain a single virtual session for each client, corresponding to the composed service, and to invalidate it together with all the client's sessions if one of its sessions terminates.

4 ZooNet

We implement *ZooNet*, a modular composition of ZooKeepers. Though in principle, modular composition requires only client-side support, we identified a design issue in ZooKeeper that makes remote learner (observer) deployments slow due to poor isolation among clients. Since remote learners are instrumental to our solution, we address this issue in the ZooKeeper server, as detailed in Section 4.1. We then discuss our client-side code in Section 4.2.

4.1 Server-Side Isolation

The original ZooKeeper implementation stalls reads when there are concurrent updates by other clients. Generally speaking, reads wait until an update is served even when the semantics do not require it. In Section 4.1.1 we describe this problem in more detail and in Section 4.1.2 we present our solution, which we have made available as a patch to ZooKeeper [21] and has been recently committed to ZooKeeper's main repository.

4.1.1 ZooKeeper's Commit Processor

ZooKeeper servers consist of several components that process requests in a pipeline. When an update request arrives to a ZooKeeper server from a client, the server forwards the update to the leader and places the request in a local queue until it hears from the leader that voting on the update is complete (i.e., the leader has *committed* the request). Only at that point the update can be applied to the local server state. A component called *commit processor* is responsible for matching incoming client requests with commit responses received from the leader, while maintaining the order of operations submitted by each client.

In the original implementation of the commit processor, (up to ZooKeeper version 3.5.1-alpha), clients are not isolated from each other: once some update request reaches the head of the request stream, all pending requests by all clients connected to this server stall until a commit message for the head request arrives from the leader. This means that there is a period, whose duration depends on the round-trip latency between the server and the leader plus the latency of quorum voting, during which all requests are stalled. While the commit processor must maintain the order of operations submitted by each client, enforcing order among updates of *different* clients is the task of the leader. Hence, blocking requests of other clients in this situation, only because they were unlucky enough to connect via the same server, is redundant.

In a geo-distributed deployment, this approach severely hampers performance as it does not allow read operations to proceed concurrently with long-distance concurrent updates. In the context of modular composition, it means that syncs hamper read-intensive workloads, i.e., learners cannot serve reads locally concurrently with syncs and updates.

4.1.2 Commit Processor Isolation

We modified ZooKeeper's commit processor to keep a separate queue of pending requests per client. Incoming reads for which there is no preceding pending update by the same client, (i.e., an update for which a commit message has not yet been received), are not blocked. Instead, they are forwarded directly to the next stage of the pipeline, which responds to the client based on the current server state.

Read requests of clients with pending updates are enqueued in the order of arrival in the appropriate queue. For each client, whenever the head of the queue is either a committed update or a read, the request is forwarded to the next stage of the server pipeline. Updates are marked committed according to the order of commit messages received from the leader (the linearization order). For more details, see our ZooKeeper Jira [21].

4.2 The ZooNet Client

We prototyped the ZooNet client as a wrapper for ZooKeeper's Java client library. It allows clients to establish sessions with multiple ZooKeeper ensembles and maintains these connections. Users specify the target ZooKeeper ensemble for every operation as a znode path prefix. Our library strips this prefix and forwards the operation to the appropriate ZooKeeper, converting some of the reads to synced reads in accordance with Algorithm 1. Our sync operation performs a dummy update; we do so because ZooKeeper's sync is not a linearizable update [28]. The client wrapper consists of roughly 150 lines of documented code.

5 Evaluation

We now evaluate our modular composition concept using the ZooNet prototype. In Section 5.1 we describe the environment in which we conduct our experiments. Section 5.2 evaluates our server-side modification to ZooKeeper, whereas Section 5.3 evaluates the cost of the synchronization introduced by ZooNet's client. Finally, Section 5.4 compares ZooNet to a single ZooKeeper ensemble configured to ensure consistency using remote learners (Figure 2b).

5.1 Environment and Configurations

We conduct our experiments on Google Compute Engine [10] in two data centers, DC1 in eastern US (South Carolina) and DC2 in central US (Iowa). In each data center we allocate five servers: three for a local ZooKeeper ensemble, one for a learner connected to the remote data center, and one for simulating clients (we run 30 request-generating client threads in each data center). Each server is allocated a standard 4 CPU machine with 4 virtual CPUs and 15 GB of memory. DC1 servers are allocated on a 2.3 GHz Intel Xeon E5 v3 (Haswell) platform, while DC2 servers are allocated on a 2.5GHz Intel Xeon E5 v2 (Ivy Bridge). Each server has two standard persistent disks. The Compute Engine does not provide us with information about available network bandwidth between the servers. We use the latest version of ZooKeeper to date, version 3.5.1-alpha.

We benchmark throughput when the system is saturated and configured as in ZooKeeper's original evaluation (Section 5.1 in [28]). We configure the servers to log requests to one disk while taking snapshots on another. Each client thread has at most 200 outstanding requests at a time. Each request consists of a read or an update of 1KB of data. The operation type and target coordination service are selected according to the workload specification in each experiment.

5.2 Server-Side Isolation

In this section we evaluate our server-side modification given in Section 4.1. We study the learner's throughput with and without our change. Recall that the learner (observer in ZooKeeper terminology) serves as a fast local read cache for distant clients, and also forwards update requests to the leader.

We experiment with a single ZooKeeper ensemble running three acceptors in DC1 and an observer in DC2. Figure 4 compares the learner's throughput with and without our modification, for a varying percentage of reads in the workload. DC1 clients have the same workload as DC2 clients.

Our results show that for read-intensive workloads that include some updates, ZooNet's learner gets up to around 4x higher throughput by allowing concurrency between reads and updates of different clients, and there is 30% up to 60% reduction in the tail latency. In a read-only workload, ZooNet does not improve the throughput or the latency, because ZooKeeper does not stall any requests. In write-intensive workloads, reads are often blocked by preceding pending updates by the same client, so few reads can benefit from our increased parallelism.

Our Jira [21] provides additional evaluation (conducted on Emulab [43]) in which we show that the throughput speedup for local clients can be up to 10x in a single data center deployment of ZooKeeper. Moreover, ZooNet significantly reduces read and write latency in mixed workloads in which the write percentage is below 30 (for reads, we get up to 96% improvement, and for writes up to 89%).

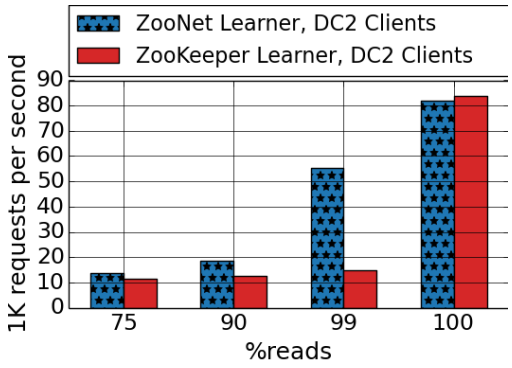


Figure 4: Improved server-side isolation. Learner's throughput as a function of the percentage of reads.

5.3 The Cost of Consistency

ZooNet is a composition of independent ZooKeepers, as depicted in Figure 2c, with added sync requests. In this section we evaluate the cost of the added syncs by comparing our algorithm to two alternatives: (1) Sync-All, where all reads are executed as synced reads, and (2) Never-Sync, in which clients never perform synced reads.

Never-Sync in not sequentially consistent (as illustrated in Figure 1). It thus corresponds to the fastest but inconsistent ZooKeeper deployment (Figure 2c), with ZooKeeper patched to improve isolation. At the other extreme, by changing all reads to be synced, Sync-All guarantees linearizability for all operations, including reads. ZooNet provides a useful middle ground (supported by most coordination services in the single-data center setting), which satisfies sequential consistency for all operations and linearizability for updates.

As a sanity check, we study in Section 5.3.1 a fully partitionable workload with clients accessing only local data in each data center. In Section 5.3.2 we have DC1 clients perform only local operations, and DC2 clients perform both local and remote operations.

5.3.1 Local Workload

In Figure 5 we depict the saturation throughput of DC1 (solid lines) and DC2 (dashed lines) with the three alternatives.

ZooNet's throughput is identical to that of Never-Sync in all workloads, at both data centers. This is because

ZooNet sends sync requests only due to changes in the targeted ZooKeeper, which do not occur in this scenario. Sync-All has the same write-only throughput (leftmost data point). But as the rate of reads increases, Sync-All performs more synced reads, resulting in a significant performance degradation (up to 6x for read-only workloads). This is because a read can be served locally by any acceptor (or learner), whereas each synced read, similarly to an update, involves communication with the leader and a quorum.

The read-only throughput of ZooNet and Never-Sync is lower than we expect: since in this scenario the three acceptors in each data center are dedicated to read requests, we would expect the throughput to be 3x that of a single learner (reported in Figure 4). We hypothesize that the throughput is lower in this case due to a network bottleneck.

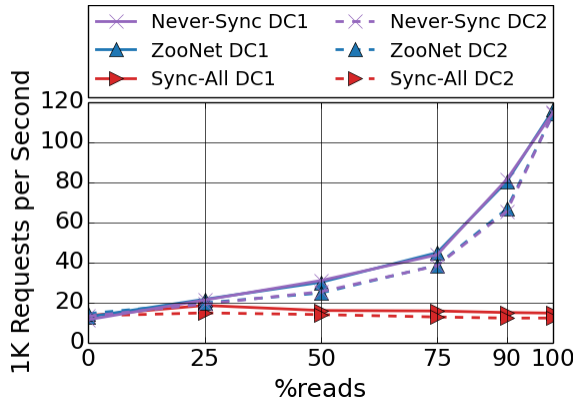


Figure 5: Saturated ZooNet throughput at two data centers with local operations only. In this sanity check we see that the performance of Never-Sync is identical to ZooNet's performance when no syncs are needed.

5.3.2 Remote Data Center Access

When clients access remote data, synced reads kick-in and affect performance. We now evaluate the cost of synced reads as a function of workload locality. We define two workload parameters: *local operations*, which represents spatial locality, namely the percentage of requests that clients address to their local data center, and *burst*, which represents the temporal locality of the target ZooKeeper. For simplicity, we consider a fixed burst size, where the client sends *burst* requests to the same ZooKeeper and then chooses a new target ZooKeeper according to the local operations ratio. Note that a burst size of 1 represents the worst-case scenario for ZooNet, while with high burst sizes, the cost of adding syncs is minimized.

Our design is optimized for partitionable workloads where spatial locality is high by definition since clients rarely access data in remote partitions. In ZooKeeper, another factor significantly contributes to temporal locality: ZooKeeper limits the size of each data object (called znode) to 1MB, which causes applications to express stored state using many znodes, organized in a hierarchical manner. ZooKeeper intentionally provides a minimalistic API, so programs wishing to access stored state (e.g., read the contents of a directory or sub-tree) usually need to make multiple read requests to ZooKeeper, effectively resulting in a high burst size.

In Figure 6 we compare ZooNet to Sync-All and Never-Sync with different burst sizes where we vary the local operations ratio of DC2 clients. DC1 clients perform 100% local operations. We select three read ratios for this comparison: a write-intensive workload in which 50% of the requests are updates (left column), a read-intensive workload in which 90% of the requests are reads (middle column), and a read-only workload (right column). DC1 clients and DC2 clients have the same read ratio in each test.

Results show that in a workload with large bursts of 25 or 50 (bottom two rows), the addition of sync requests has virtually no effect on throughput, which is identical to that of Never-Sync except in read-intensive workloads, where with a burst of 25 there is a slight throughput degradation when the workload is less than 80% local.

When there is no temporal locality (burst of 1, top row), the added syncs induce a high performance cost in scenarios with low spatial locality, since they effectively modify the workload to become write-intensive. In case most accesses are local, ZooNet seldom adds syncs, and so it performs as well as Never-Sync regardless of the burst size.

All in all, ZooNet incurs a noticeable synchronization cost only if the workload shows no locality whatsoever, neither temporal nor spatial. Either type of locality mitigates this cost.

5.4 Comparing ZooNet with ZooKeeper

We compare ZooNet with the fastest cross data center deployment of ZooKeeper that is also consistent, i.e., a single ZooKeeper ensemble where all acceptors are in DC1 and a learner is located in DC2 (Figure 2b). The single coordination service deployment (Figure 2a) is less efficient since: (1) acceptors participate in the voting along with serving clients (or, alternatively, more servers need to be deployed as learners as in [41]); and (2) the voting is done over WAN (see [13] for more details). We patch ZooKeeper with the improvement described in Section 4.1 and set the burst size to 50 in order to focus the current discussion on the impact that data locality has on performance.

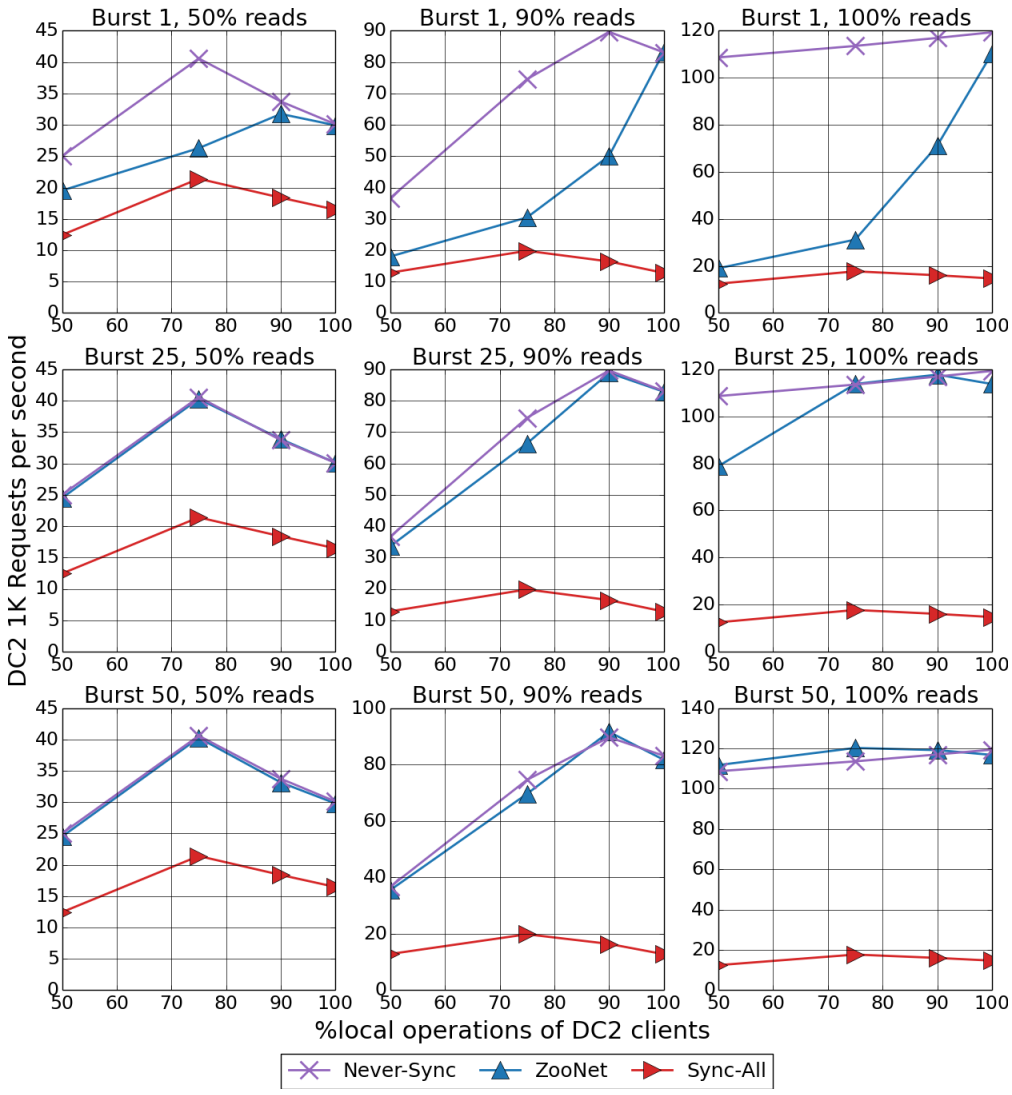


Figure 6: Throughput of ZooNet, Never-Sync and Sync-All. Only DC2 clients perform remote operations.

We measure aggregate client throughput and latency in DC1 and DC2 with ZooKeeper and ZooNet, varying the workload's read ratio and the fraction of local operations of the clients in DC2. We first run a test where all operations of clients in DC1 are local. Figure 7a shows the throughput speedup of ZooNet over ZooKeeper at DC1 clients, and Figure 7b shows the throughput speedup for DC2 clients.

Our results show that in write-intensive workloads, DC2 clients get up to 7x higher throughput and up to 92% reduction in latency. This is due to the locality of update requests in ZooNet, compared to the ZooKeeper deployment in which each update request of a DC2 client is forwarded to DC1. The peak throughput saturates at the update rate that a single leader can handle. Beyond that saturation point, it is preferable to send update operations to a remote DC rather than have them handled locally, which leads to a decrease in total throughput.

In read-intensive workloads (90% – 99% reads), DC2 clients also get a higher throughput with ZooNet (4x to 2x), and up to 90% reduction in latency. This is due to the fact that in ZooKeeper, a single learner can handle a lower update throughput than three acceptors. In read-only workloads, the added acceptors have less impact on throughput; we assume that this is due to a network bottleneck as observed in our sanity check above (Figure 5).

In addition, we see that DC1 clients are almost unaffected by DC2 clients in read-intensive workloads. This is due to the fact that with both ZooKeeper and ZooNet, reads issued by clients in DC2 are handled locally in DC2. The added synced reads add negligible load to the acceptors in DC1 due to the high burst size and locality

of requests (nevertheless, they do cause the throughput speedup to drop slightly below 1 when there is low locality). With a write-intensive workload, DC1 clients have a 1.7x throughput speedup when DC2 clients perform no remote operations. This is because remote updates of DC2 clients in ZooKeeper add to the load of acceptors in DC1, whereas in ZooNet some of these updates are local and processed by acceptors in DC2.

Finally, we examine a scenario where clients in both locations perform remote operations. Figure 8a shows the throughput speedup of ZooNet over ZooKeeper achieved at DC1 clients, and Figure 8b shows the throughput speedup of DC2 clients. All clients have the same locality ratio. Each curve corresponds to a different percentage of reads.

There are two differences between the results in Figure 8 and Figure 7. First, up to a local operations ratio of 75%, DC1 clients suffer from performance degradation in read-intensive workloads. This is because in the ZooKeeper deployment, all the requests of DC1 clients are served locally, whereas ZooNet serves many of them remotely. This re-emphasizes the observation that ZooNet is most appropriate for scenarios that exhibit locality, and is not optimal otherwise.

Second, the DC1 leader is less loaded when DC1 clients also perform remote updates (Figure 8). This mostly affects write-intensive scenarios (top blue curve), in which the leaders at both data centers share the update load, leading to higher throughput for all clients. Indeed, this yields higher throughput speedup when locality is low (leftmost data point in Figures 8a and 8b compared to Figures 7a and 7b, respectively). As locality increases to 70%–80%, the DC2 leader becomes more loaded due to DC2s updates, making the throughput speedup in Figures 7b and Figure 8b almost the same, until with 100% local updates (rightmost data point), the scenarios are identical and so is the throughput speedup.

6 Related Work

Coordination services such as ZooKeeper [28], Chubby [24], etcd [9], and Consul [5] are extensively used in industry. Many companies deploying these services run applications in multiple data centers. But questions on how to use coordination services in a mutli-data center setting arise very frequently [4, 11, 15, 16, 17], and it is now clear that the designers of coordination services must address this use-case from the outset.

In what follows we first describe the current deployment options in Section 6.1 followed by a discussion of previously proposed composition methods in Section 6.2.

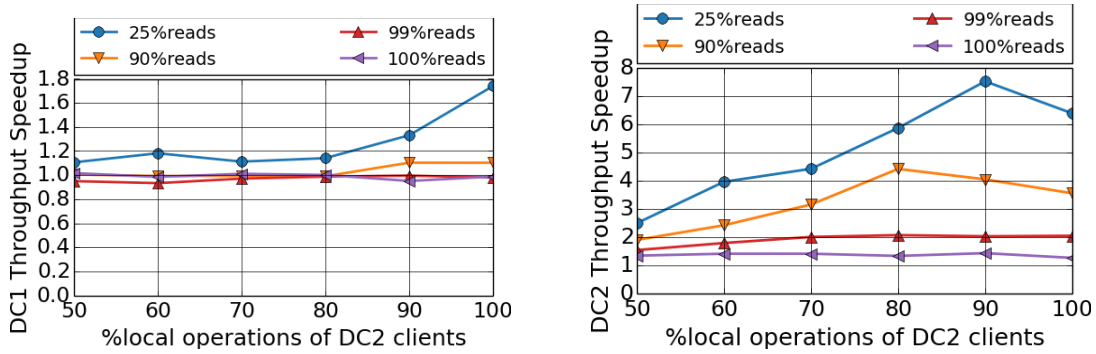
A large body of work, e.g., [30, 34, 35], focuses on improving the efficiency of coordination services. Our work is orthogonal – it allows combining multiple instances to achieve a single system abstraction with the same semantics, while only paying for coordination when it is needed.

6.1 Multi-Data Center Deployment

In Section 2 we listed three prevalent strategies for deploying coordination services across multiple data centers: a single coordination service where acceptors are placed in multiple data centers, a single coordination service where acceptors run in one data center, or multiple coordination services. The choice among these options corresponds to the tradeoff system architects make along three axes: consistency, availability, and performance (a common interpretation of the CAP theorem [7]). Some are willing to sacrifice update speed for consistency and high-availability in the presence of data center failures [22, 25, 40, 41]. Others prefer to trade-off fault-tolerance for update speed [13], while others prioritize update speed over consistency [4, 11]. In this work we mitigate this tradeoff, and offer a fourth deployment option whose performance and availability are close to that of the third (inconsistent) option, without sacrificing consistency.

Some systems combine more than one of the deployment alternatives described in Section 2. For example, Vitess [20] deploys multiple local ZooKeeper ensembles (as in Figure 2c) in addition to a single global ensemble (as in Figure 2a). The global ensemble is used to store global data that doesn't change very often and needs to survive a data center failure. A similar proposal has been made in the context of SmartStack, Airbnb's service discovery system [12]. ZooNet can be used as-is to combine the local and global ensembles in a consistent manner.

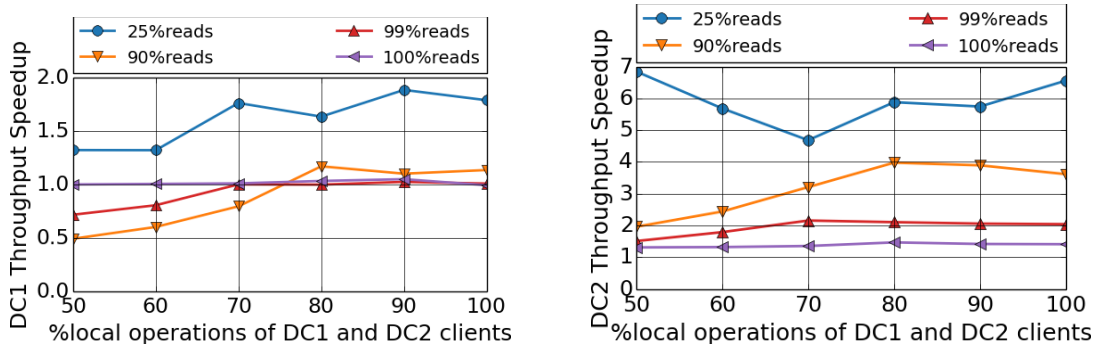
Multiple studies [38, 44] showed that configuration errors and in particular inconsistencies are a major source of failure for Internet services. To prevent inconsistencies, configuration stores often use strongly consistent coordination services. ACMS [40] is Akamai's distributed configuration store, which, similarly to Facebook's Zeus [41], is based on a single instance of a strongly consistent coordination protocol. Our design offers a scalable alternative where, assuming that the stored information is highly partitionable, updates rarely go through WAN and can execute with low latency and completely independently in the different partitions, while all reads (even of data stored remotely) remain local. We demonstrate that the amortized cost of sync messages is low for such read-heavy systems (in both ACMS and Zeus the reported rate of updates is only hundreds per hour).



(a) Throughput speedup of DC1 clients.

(b) Throughput speedup of DC2 clients.

Figure 7: Throughput speedup (ZooNet/ZooKeeper). DC1 clients perform only local operations. The percentage of read operations is identical for DC1 clients and DC2 clients.



(a) Throughput speedup of DC1 clients.

(b) Throughput speedup of DC2 clients.

Figure 8: Throughput speedup (ZooNet/ZooKeeper). DC1 clients an DC2 clients have the same local operations ratio as well as read operations percentage.

6.2 Composition Methods

Consul [5], ZooFence [26] and Volery [23] are coordination services designed with the multi-data center deployment in mind. They provide linearizable updates and either linearizable or sequentially consistent reads. Generally, these systems follow the multiple coordination services methodology (Figure 2c) – each coordination service is responsible for part of the data, and requests are forwarded to the appropriate coordination service (or to a local proxy). As explained in Section 2, when the forwarded operations are sequentially-consistent reads, this method does not preserve the single coordination service's semantics. We believe that, as in ZooKeeper, this issue can be rectified using our modular composition approach.

ZooFence [26] orchestrates multiple instances of ZooKeeper using a client-side library in addition to a routing layer consisting of replicated queues and executors. Intuitively, it manages local and cross-data center partitions using data replication. Any operation (including reads) accessing replicated data must go through ZooFence's routing layer. This prevents reads from executing locally, forfeiting a major benefit of replication. In contrast, ZooNet uses learners, (which natively exist in most coordination services in the form or proxies or observers), for data replication. This allows local reads, and does not require orchestration of multiple ZooKeeper instances as in ZooFence.

Volery [23] is an application that implements ZooKeeper's API, and which consists of partitions, each of which is an instance of a state machine replication algorithm. Unlike ZooKeeper, all of Volery operations are linearizable (i.e., including reads). In Volery, the different partitions must communicate among themselves in order to maintain consistency, unlike ZooNet's design in which the burden of maintaining consistency among ZooKeepers is placed only on clients. In addition, when compared to ZooKeeper, Volery shows degredated performance in case of a single partition, while ZooNet is identical to ZooKeeper if no remote operations are needed.

In distributed database systems, composing multiple partitions is usually done with protocols such as two-phase commit (e.g., as in [25]). In contrast, all coordination services we are familiar with are built on key-value stores, and expose simpler non-transactional updates and reads supporting non-ACID semantics.

Server-side solutions were also proposed for coordination services composition [14] but were never fully implemented due to their complexity, the intrusive changes they require from the underlying system, as well as the proposed relaxation of coordination service's semantics required to make them work. In this paper we show that composing such services does not require expensive server-side locking and commit protocols among partitions, but rather can be done using a simple modification of the client-side library and can guarantee the standard coordination service semantics.

7 Conclusions and Future Work

Coordination services provide consistent and highly available functionality to applications, relieving them of implementing common (but subtle) distributed algorithms on their own. Yet today, when applications are deployed in multiple data centers, system architects are forced to choose between consistency and performance. In this paper we now shown that this does not have to be the case. Our modular composition approach maintains the performance and simplicity of deploying independent coordination services in each data center, and yet does not forfeit consistency.

We demonstrated that the simplicity of our technique makes it easy to use with existing coordination services, such as ZooKeeper – it does not require changes to the underlying system, and existing clients may continue to work with an individual coordination service without any changes (even if our client library is used, such applications will not incur any overhead). Moreover, the cost for applications requiring consistent multi-data center coordination is low for workloads that exhibit high spatial or temporal locality.

In this work we have focused on the advantages of our composition design in wide-area deployments. It is possible to leverage the same design for deployments within the data center boundaries that currently suffer from lack of sharing among coordination services. Indeed, a typical data center today runs a multitude of coordination service backend services. For example, it may include: Apache Kafka message queues [2], backed by ZooKeeper and used in several applications; Swarm [19], a Docker [36] clustering system running an etcd backend; Apache Solr search platform [3] with an embedded ZooKeeper instance; and Apache Storm clusters [42], each using a dedicated ZooKeeper instance. Thus, installations end up running many independent coordination service instances, which need to be independently provisioned and maintained. This has a number of drawbacks: (1) it does not support cross-application sharing; (2) it is resource-wasteful, and (3) it complicates system administration. Our modular composition approach can potentially remedy these short comings.

Our composition algorithm supports individual query and update operations. It can natively support transactions (e.g., ZooKeeper's *multi* operation) involving data in single service instance. An interesting future direction could be to support transactions involving multiple service instances. This is especially challenging in the face of possible client and service failures, if all cross-service coordination is to remain at the client side.

Acknowledgements

We thank Arif Merchant, Mustafa Uysal, John Wilkes, and the anonymous reviewers for helpful comments and suggestions. We gratefully acknowledge Google for funding our experiments on Google Cloud Platform. We thank Emulab for the opportunity to use their testbeds. Kfir Lev-Ari is supported in part by the Hasso-Plattner Institute (HPI) Research School. Research partially done while Kfir Lev-Ari was an intern with Yahoo, Haifa. Partially supported by the Israeli Ministry of Science.

References

[1] Access Control in Google Cloud Storage. `https://cloud.google.com/storage/docs/access-control`. [Online; accessed 28-Jan-2016].

[2] Apache Kafka – A high-throughput distributed messaging system. `http://kafka.apache.org`. [Online; accessed 1-Jan-2016].

[3] Apache Solr – a standalone enterprise search server with a REST-like API. `http://lucene.apache.org/solr/`. [Online; accessed 1-Jan-2016].

[4] Camille Fournier: Building a Global, Highly Available Service Discovery Infrastructure with ZooKeeper. `http://whilefalse.blogspot.co.il/2012/12/building-global-highly-available.html`. [Online; accessed 1-Jan-2016].

[5] Consul – a tool for service discovery and configuration. Consul is distributed, highly available, and extremely scalable. `https://www.consul.io/`. [Online; accessed 1-Jan-2016].

[6] Consul HTTP API. `https://www.consul.io/docs/agent/http.html`. [Online; accessed 28-Jan-2016].

[7] Daniel Abadi: Problems with CAP, and Yahoos little known NoSQL system. `http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html`. [Online; accessed 28-Jan-2016].

[8] Doozer – a highly-available, completely consistent store for small amounts of extremely important data. `https://github.com/ha/doozerd`. [Online; accessed 1-Jan-2016].

[9] etcd – a highly-available key value store for shared configuration and service discovery. https://coreos.com/etcd/. [Online; accessed 1-Jan-2016].

[10] Google Compute Engine – Scalable, High-Performance Virtual Machines. https://cloud.google.com/compute/. [Online; accessed 1-Jan-2016].

[11] Has anyone deployed a ZooKeeper ensemble across data centers? https://www.quora.com/Has\anyone\deployed\a\ZooKeeper\ensemble\across\data\centers. [Online; accessed 1-Jan-2016].

[12] Igor Serebryany: SmartStack vs. Consul. http://igor.moomers.org/smartstack-vs-consul/. [Online; accessed 28-Jan-2016].

[13] Observers: Making ZooKeeper Scale Even Further. https://blog.cloudera.com/blog/2009/12/observers-making-zookeeper-scale-even-further/. [Online; accessed 1-Jan-2016].

[14] Proposal: mounting a remote ZooKeeper. https://wiki.apache.org/hadoop/ZooKeeper/MountRemoteZookeeper. [Online; accessed 28-Jan-2016].

[15] Question about cross-datacenter setup (ZooKeeper). http://goo.gl/0sDOMZ. [Online; accessed 28-Jan-2016].

[16] Question about multi-datacenter key-value consistency (Consul). https://goo.gl/XMWCcH. [Online; accessed 28-Jan-2016].

[17] Question about number of nodes spread across datacenters (ZooKeeper). https://goo.gl/oPC2Yf. [Online; accessed 28-Jan-2016].

[18] Solr Cross Data Center Replication. http://yonik.com/solr-cross-data-center-replication/. [Online; accessed 28-Jan-2016].

[19] Swarm: a Docker-native clustering system. https://github.com/docker/swarm. [Online; accessed 1-Jan-2016].

[20] Vitess deployment: global vs local. http://vitess.io/doc/TopologyService/#global-vs-local. [Online; accessed 28-Jan-2016].

[21] ZooKeeper's Jira - Major throughput improvement with mixed workloads. https://issues.apache.org/jira/browse/ZOOKEEPER-2024. [Online; accessed 16-May-2016].

[22] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[23] BEZERRA, C. E. B., PEDONE, F., AND VAN RENESSE, R. Scalable state-machine replication. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014* (2014), pp. 331–342.

[24] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[25] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[26] HALALAI, R., SUTRA, P., RIVIERE, E., AND FELBER, P. Zoofence: Principled service partitioning and application to the zookeeper coordination service. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014* (2014), pp. 67–78.

[27] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[28] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2010), USENIXATC'10, USENIX Association, pp. 11–11.

[29] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks* (Washington, DC, USA, 2011), DSN '11, IEEE Computer Society, pp. 245–256.

[30] KAPRITSOS, M., WANG, Y., QUEMA, V., CLEMENT, A., ALVISI, L., AND DAHLIN, M. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 237–250.

[31] LAMPORT, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28*, 9 (Sept. 1979), 690–691.

[32] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998), 133–169.

[33] LEV-ARI, K., BORTNIKOV, E., KEIDAR, I., AND SHRAER, A. Modular composition of coordination services. Tech. Rep. CCIT 895, EE, Technion, Januar 2016.

[34] MAO, Y., JUNQUEIRA, F. P., AND MARZULLO, K. Mencius: Building efficient replicated state machine for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings* (2008), pp. 369–384.

[35] MARANDI, P. J., BEZERRA, C. E., AND PEDONE, F. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems* (Washington, DC, USA, 2014), ICDCS '14, IEEE Computer Society, pp. 368–377.

[36] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. *Linux J. 2014*, 239 (Mar. 2014).

[37] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[38] OPPENHEIMER, D. L., GANAPATHI, A., AND PATTERSON, D. A. Why do internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems, USITS'03, Seattle, Washington, USA, March 26-28, 2003* (2003).

[39] SHAROV, A., SHRAER, A., MERCHANT, A., AND STOKELY, M. Take me to your leader!: Online optimization of distributed storage configurations. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1490–1501.

[40] SHERMAN, A., LISIECKI, P. A., BERKHEIMER, A., AND WEIN, J. ACMS: the akamai configuration management system. In *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings.* (2005).

[41] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 328–343.

[42] TOSHNIWAL, A., TANEJA, S., SHUKLA, A., RAMASAMY, K., PATEL, J. M., KULKARNI, S., JACKSON, J., GADE, K., FU, M., DONHAM, J., BHAGAT, N., MITTAL, S., AND RYABOY, D. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 147–156.

[43] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation* (Boston, MA, Dec. 2002), USENIX Association, pp. 255–270.

[44] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 159–172.

Cheap and Available State Machine Replication

Rong Shi
The Ohio State University

Yang Wang
The Ohio State University

Abstract

This paper presents that, by combining on-demand instantiation and lazy recovery, we can reduce the cost of asynchronous state machine replication protocols, such as Paxos and UpRight, while maintaining their high availability. To reduce cost, we incorporate on-demand instantiation, which activates a subset of replicas first and activates backup ones when active ones fail. To solve its key limitation—the system can be halted for long when activating a backup replica, we apply lazy recovery, allowing the system to proceed while recovering backup nodes in the background. The key contribution of this paper is to identify that, when agreement nodes and execution nodes are logically separated, they each presents a unique property that enables lazy recovery. We have applied this idea to Paxos and built ThriftyPaxos, which, as shown in the evaluation, can achieve higher throughput and similar availability comparing to standard Paxos, despite the fact that ThriftyPaxos activates fewer replicas.

1 Introduction

This paper presents that, by combining on-demand instantiation [37, 59] and lazy recovery [31, 32], we can reduce the cost of asynchronous state machine replication (SMR) protocols [52], such as Paxos [33, 34, 48] and UpRight [18], while maintaining their high availability.

Replication is widely used in today's storage systems to protect data against failures. In general, stronger replication protocols that can tolerate more kinds of errors usually need more replicas. For example, primary backup protocols [11, 13, 23, 57] can tolerate f machine crashes with $f+1$ replicas, which is the minimal one can expect. Paxos [33, 34, 48] needs $2f+1$ replicas to tolerate f machine crashes and asynchronous events (e.g inaccurate timeout caused by network partitions or slow machines). To further tolerate arbitrary failures, Byzantine Fault Tolerance (BFT) protocols [1, 6, 15, 18, 20, 30, 38] need at least $3f+1$ replicas.

Requiring more replicas incurs a higher cost: the system needs more storage space to store data, more bandwidth to transfer data, and more processors to process data. Such additional cost is magnified in today's large data centers, where up to millions of machines may need to be replicated [19].

Whether to pay such additional cost for stronger guarantees becomes a hard question for developers. Indeed, while a number of systems are using Paxos to replicate their data [3–5, 10, 14, 19], many others are still using primary backup or similar protocols [23, 26, 47, 53], which, for consistency, have to use a conservative (long) timeout to reduce the possibility of asynchronous events. Conservative timeout, however, can hurt system availability when replicas fail.

Existing attempts to reduce replication cost are only effective for certain applications or protocols. For example, separating agreement from execution [60] can reduce the number of execution replicas in BFT protocols to $2f+1$, but it is not effective for applications whose agreement is the bottleneck or those that are using Paxos. On-demand instantiation [37, 59] activates the minimal number of replicas first, and activates backup ones when the active ones fail. However, before a backup replica can take its responsibility, it must transfer the current state from an active replica, and the system is unavailable during state transfer: for applications with a large state, the system can be halted for long. Gnothi [58] separates data from metadata, performs partial replication for data, and performs full replication for metadata: it works effectively for applications whose data is much larger than metadata, but may not work efficiently for others.

This paper, instead, presents a general approach to reduce the replication cost of asynchronous SMR protocols, while maintaining their availability properties. To reduce replication cost, our approach incorporates the idea of on-demand instantiation, which activates a subset of replicas first and activates backup ones when active

ones fail. To address its key limitation—the system may be unavailable for a long time when a backup replica is rebuilding its state, our approach incorporates lazy recovery [31, 32], which rebuilds a backup replica's state in the background without halting the system.

Neither on-demand instantiation nor lazy recovery is novel. The key contribution of this work is to identify that, for SMR protocols, *lazy recovery is possible only when agreement and execution are separated*.

SMR protocols first run an agreement protocol across replicas to decide the next request to execute and then execute the request on each replica. Therefore, a replica can be logically separated into an agreement node, which runs the agreement protocol, and an execution node, which runs the application's logic to execute the request. While previous works have exploited such separation for either clarifying protocols [33, 34] or calculating theoretical bounds [18, 60], this paper exploits the same idea for a different purpose: when an agreement node and an execution node are separated, they each presents a unique property that enables lazy recovery:

Instant activation for agreement. In principle, an agreement protocol needs to answer the question "what is the next request to execute". This question has the "memoryless" property that an agreement node does not need to know prior requests to decide the next one. This suggests that when an active agreement node fails, a blank agreement node can join the protocol instantly.

Separating critical and flexible tasks for execution. An execution node must execute requests in order, because execution of later requests may depend on information in earlier requests. Its opportunity for lazy recovery comes from a different property: critical latency-sensitive tasks that must be performed for availability (e.g. executing a request and replying to the client) sometimes require fewer replicas than flexible background tasks that can be delayed (e.g. garbage collection). For example, in Paxos, a client needs only one reply from any replica to proceed, but a garbage collection requires $f+1$ replicas to take a snapshot. While existing protocols try to ensure that, despite failures, the system has enough replicas to execute even flexible tasks, this may not be necessary: activating an appropriate number of replicas to ensure the availability of critical tasks and relying on lazy recovery for flexible tasks may achieve both low cost and high availability.

These properties enable lazy recovery for both agreement and execution, but in different ways: when an active agreement node fails, a blank backup agreement node can join the protocol instantly; when an active execution node fails, the system can proceed with remaining active execution nodes (they execute critical latency-sensitive tasks but delay flexible background tasks). In both cases, the system rebuilds the state of a backup node in the background.

This paper formally studies the number of active nodes required for availability. Here we highlight some results:

- For Paxos, we need $f+1$ active agreement nodes and active execution nodes.
- For BFT, when setting distinct bounds for omission failures (u) and commission failures (r) [18] instead of setting a unified f for both, we need $u+r+1$ active execution nodes. This is smaller than the previous $2f+1$ lower bound, in a practical setting when $u > r$.

In order for our approach to work properly, there is one additional challenge we need to address: although recovery of backup nodes can be delayed, recovery still has to be completed in a timely manner. Otherwise, long recovery can hurt the durability of the system. Furthermore, delaying flexible tasks like garbage collection for too long can eventually block the system. Recovery is further complicated by the fact that it is performed in parallel with executing new requests and they often compete for resource. To address these challenges, we introduce an *adaptive recovery* mechanism, which allows a user to specify a soft deadline for recovery: our mechanism makes best effort to meet the deadline while using the remaining resource to process new requests. To achieve this, it continuously monitors the progress of recovery and adaptively adjusts resource allocation.

We have applied this idea to Paxos, a popular replication protocol in today's datacenters, to build ThriftyPaxos, which can achieve the same correctness guarantee as Paxos with f fewer replicas. Our evaluation shows that ThriftyPaxos can achieve higher throughput and similar availability comparing to Paxos, despite the fact that ThriftyPaxos activates fewer replicas.

2 Background

State machine replication (SMR) models an application as a deterministic state machine. For fault tolerance, SMR deploys multiple replicas of an application's state machine on different machines. To ensure all replicas are identical, SMR protocols run an agreement or consensus protocol across replicas to decide the next request to execute. These protocols can guarantee that, despite failures, all correct replicas will reach the same decision.

To tolerate network failures, replicas need to log requests during agreement, so that if a request is lost, the corresponding replica can retrieve it from the log and retransmit it. To ensure logs do not grow arbitrarily, SMR protocols periodically require application's state machines to take snapshots of their states, promising that

they will never need earlier requests. The system can then garbage collect log entries before the snapshot.

Previous works exploit features of SMR protocols to reduce their cost.

On-demand instantiation. Most SMR protocols (e.g. Paxos, PBFT, UpRight, etc) are designed for an *asynchronous* environment where message delivery can be delayed arbitrarily and clocks of machines can drift arbitrarily because of asynchronous events such as network partitions or machine overloading. SMR protocols are designed to be safe (all correct replicas process the same sequence of requests) despite failures and to be live (requests eventually get processed) when message delivery is timely and clocks are reasonably synchronized.

While a replicated system needs a minimal number of $f+1$ replicas to tolerate f failures, an asynchronous system needs more replicas, because in an asynchronous environment, it is impossible to accurately know whether a replica has failed or not. In this case, if the system has only $f+1$ replicas, and if one of them is not responding, it is impossible for the system to decide how to proceed: it is inappropriate to proceed with remaining ones because the unresponsive replica may just be temporarily slow and in this case, the request has not been executed by sufficient number of replicas; it is also inappropriate to wait for the unresponsive replica, because the replica may have actually failed and waiting may take for ever.

To solve this problem, asynchronous SMR protocols incorporate more replicas and only expect a subset of them to respond. Such design motivates the idea of on-demand instantiation [37, 59]: since the system needs only a subset of replicas to respond, we can activate the subset first. If all of them respond in time, the system can make progress; if some of them become unresponsive, we can activate the backup ones. Since machine failures and asynchronous events are rare, this approach can reduce the replication cost in most of the time.

On-demand instantiation can achieve the same safety and liveness properties as the original approach, because asynchronous SMR protocols are designed for an environment where some of the replicas can be arbitrarily slow: in such an environment, the on-demand instantiation approach, which disables a subset of nodes, is fundamentally indistinguishable from a special case of the original approach, in which the same subset of nodes are just slow. However, as mentioned in Section 1, previous works that adopt this idea suffer from the availability problem that a backup replica may take a long time to recover before it can function.

Separating agreement from execution. In SMR protocols, a replica can be logically separated into an agreement node, which participates in agreement, and an execution node, which runs the application's state machine.

Paxos made such separation when describing its protocol (agreement node and execution node are called acceptor and learner, respectively, in Paxos). Yin et al. observe that for BFT protocols, the number of execution nodes can be reduced to lower system cost [60]. UpRight further refines this observation [18]. However, as mentioned in Section 1, this approach is not much helpful to light applications whose agreement is the bottleneck; it is also not effective to Paxos-like protocols.

3 Combine on-demand instantiation and lazy recovery

Our approach incorporates on-demand instantiation to reduce replication cost, and addresses its availability problem by lazy recovery: when an active replica fails, the system keeps processing requests while recovering a backup replica in the background. Such combination can achieve both low cost and high availability.

To incorporate lazy recovery, however, we must ensure that the system is able to function correctly even when part of the system is in recovery and thus only has partial state—this is the major challenge of this work. The key contribution of this paper is to identify that lazy recovery is possible only when agreement node and execution nodes are logically separated. When separated, they each presents a unique property that enables lazy recovery.

3.1 Instant activation for agreement node

An agreement protocol needs to decide the next request to execute, and this task has the memoryless property that an agreement node does not need to know prior requests to decide the next one. Such memoryless property allows a blank backup agreement node to join the protocol instantly when an active one becomes unresponsive.

Number of active nodes for agreement. Suppose an SMR protocol needs a maximum number of N^A_{max} agreement nodes, in which f of them can fail. Also suppose that the SMR protocol needs N^A_{normal} agreement nodes to participate in agreement in the failure-free case. In most SMR protocols, $N^A_{normal} = N^A_{max} - f$, because there is no guarantee that more nodes can respond. However, some protocols, such as Fast Paxos [35] and Zyzzyva [30], introduce a fast path, which requires more than $N^A_{max} - f$ replicas to respond, to decide the next request with less latency. When the fast path is not possible, these protocols resolve back to traditional approaches. For these protocols, N^A_{normal} could be larger than $N^A_{max} - f$.

Because a backup node can join agreement instantly, the system should activate only N^A_{normal} agreement nodes. The safety and liveness of this approach is the same as the original approach, as discussed in Section 2.

Availability. As long as the N^A_{normal} agreement nodes are correct and can communicate with each other, our system can process requests correctly. When an agreement node becomes unresponsive, our system activates a backup node. To detect failures quickly, we can use aggressive techniques (Section 6), because asynchronous replication does not rely on the accuracy of failure detection for correctness. The activation only takes a few messages. Therefore, the system will not halt for long when an agreement node becomes unresponsive.

To avoid frequent conflict (different agreement nodes propose different requests), many agreement protocols elect one node as the leader to propose the next request. When the leader fails, the system may halt for a while to elect a new leader and rebuild its state. Both the original protocols and our approach have to pay such cost.

3.2 Separating critical and flexible tasks for execution node

The key observation that enables lazy recovery for execution nodes is that the number of replicas required to execute critical tasks (e.g. executing a request) is sometimes fewer than that required to execute flexible tasks (e.g. garbage collection). On the one hand, the system should activate sufficient number of nodes so that, despite failures, the system can always process critical tasks; on the other hand, it does not need to be so conservative for flexible tasks because they can be delayed.

Number of active nodes for execution. Suppose an SMR protocol needs $N^E_{critical}$ execution nodes to perform critical tasks and $N^E_{flexible}$ nodes to perform flexible tasks. By following the previous idea, the system should activate $max(N^E_{critical}+f, N^E_{flexible})$ execution nodes. Once again, the safety and liveness of this approach is the same as the original approach, as discussed in Section 2.

Availability. When all active execution nodes are correct and can communicate with each other, they can perform all tasks. When no more than f active nodes are unresponsive, the system still has enough replicas to perform critical tasks. Therefore, the system can rebuild backup execution nodes in the background without halting the system. Of course, the system may not be able to perform flexible tasks until backup replicas are rebuilt.

3.3 Case studies

Table 1 shows the effectiveness of our approach when applied to popular protocols.

Paxos. The standard Paxos protocol needs a maximum of $2f+1$ replicas to tolerate asynchronous events and f crash failures. Its agreement protocol sends requests to all replicas and requires $f+1$ of them to respond; its execution requires only one execution node to reply to the client (critical task) while requiring $f+1$ execution nodes to perform a snapshot for garbage collection (flexible task). By using the previous calculations, our approach needs to activate $f+1$ agreement nodes and execution nodes.

Fast Paxos and Speculative Paxos [51] introduce a fast path, which requires more than $f+1$ agreement nodes to participate[1], to commit requests with fewer rounds of message exchanges. For these protocols, our approach needs to activate more than $f+1$ agreement nodes.

Cheap Paxos [37] applies on-demand instantiation to Paxos. It requires the same number of replicas as our approach, but since it does not incorporate lazy recovery, it suffers from the availability problem.

BFT. Practical Byzantine Fault Tolerance (PBFT) [15] needs a maximum of $3f+1$ replicas to tolerate f arbitrary failures. Its agreement protocol sends requests to all replicas and requires $2f+1$ of them to respond; its execution requires $f+1$ execution nodes to reply to the client (critical task) while requiring also $f+1$ execution nodes to perform a snapshot for garbage collection (flexible task). By using the previous calculation, our approach needs to activate $2f+1$ agreement nodes and execution nodes.

Yin et al. observe that when agreement and execution are separated, BFT protocols need only $2f+1$ execution nodes [60]. Our approach can reduce its agreement cost, but cannot reduce its execution cost. Zyzzyva [30] introduces a fast path to commit requests with less latency, but since it needs all $3f+1$ agreement nodes to respond for the fast path, our approach cannot reduce its cost.

ZZ [59] applies on-demand instantiation to BFT protocols. It requires only $f+1$ execution replicas, which is even fewer than that of our approach, but it also suffers from the availability problem. Our approach, instead, tries to minimize number of active replicas without hurting availability.

UpRight. UpRight makes a distinction between failures that can cause the system to become unavailable (its number is represented by u) and failures that can cause the system to become incorrect (its number is represented by r). Its conclusion is that we need a maximum of $u+r+max(u,r)+1$ agreement nodes and $r+max(u,r)+1$ of them should respond in the agreement protocol[2]; we need a maximum of $u+max(u,r)+1$

[1] Fast Paxos can be configured in two ways: it can be configured to have $2f+1$ agreement nodes and $f+\lfloor\frac{f}{2}\rfloor+1$ of them must respond for the fast path; it can also be configured to have $3f+1$ agreement nodes and $f+1$ of them must respond. We use the first configuration in the paper for a fair comparison with other Paxos-like protocols.

[2] UpRight further separates agreement into two phases: authentication phase needs a maximum of $u+r+max(u,r)+1$ nodes and order phase needs a maximum of $2u+r+1$ nodes. We only present the larger number in the paper for simplicity.

Protocol	Agreement		Execution			
	N^A_{max}	$N^A_{normal} = N^A_{active}$	N^E_{max}	$N^E_{critical}$	$N^E_{flexible}$	N^E_{active}
Paxos	$2f+1$	$f+1$	$2f+1$	1	$f+1$	$f+1$
Fast Paxos	$2f+1$	$f+\lfloor\frac{f}{2}\rfloor+1$	$2f+1$	1	$f+1$	$f+1$
PBFT	$3f+1$	$2f+1$	$3f+1$	$f+1$	$f+1$	$2f+1$
Yin et al.	$3f+1$	$2f+1$	$2f+1$	$f+1$	$f+1$	$2f+1$
Zyzzyva	$3f+1$	$3f+1$	$2f+1$	$f+1$	$f+1$	$2f+1$
UpRight	$u+r+max(u,r)+1$	$r+max(u,r)+1$	$u+max(u,r)+1$	$r+1$	$max(u,r)+1$	$u+r+1$

Table 1: Required number of replicas for different protocols. N^A_{max}: max number of agreement nodes; N^A_{normal}: number of agreement nodes in failure-free case; N^A_{active}: number of active agreement nodes in our approach; N^E_{max}: max number of execution nodes; $N^E_{critical}$: number of execution nodes for critical tasks; $N^E_{flexible}$: number of execution nodes for flexible tasks; N^E_{active}: number of active execution nodes in our approach.

execution nodes and $r+1$ of them should reply to the clients (critical task) while $max(u,r)+1$ of them should perform snapshots for garbage collection (flexible task). Both standard Paxos ($u=f, r=0$) and BFT ($u=r=f$) can be viewed as an instance of UpRight.

By using previous calculations ($f=u$), our approach needs to activate $r+max(u,r)+1$ agreement nodes and $u+r+1$ execution nodes. Comparing to original UpRight, our approach can always reduce its agreement cost and can reduce its execution cost when $u>r$. This conclusion shows that, when $u>r$, setting distinct u and r can reduce replication cost comparing to using a unified $f=u$. This is appealing because in most environments, the possibility of crash failures is indeed much higher than that of those Byzantine failures, indicating $u>r$ is a practical setting. Note that such opportunity to reduce execution cost by setting distinct u and r does not exist in the original UpRight protocol, because its execution cost $u+max(u,r)+1$ is equal to $2u+1$ when $u>r$ and it is not different from the $2f+1$ bound of early work [60].

4 Adaptive recovery

Although recovery can be delayed, it has to be performed in a timely manner for two reasons: first, data durability is determined by how frequently machines fail and how fast they can recover. Therefore, increasing recovery time may lead to higher probability of data loss; second, flexible tasks, such as garbage collection, cannot be performed until recovery is complete. If they are delayed for too long, eventually the system will be blocked.

In our approach, ensuring recovery speed is further complicated by the fact that recovery is performed in parallel with executing requests and these two tasks often compete for resource (e.g. network and disk bandwidth).

To complete recovery in a timely manner and to make a balance between recovery and executing new requests, this paper introduces an adaptive recovery mechanism. It allows the administrator to specify a soft deadline for recovery, which is determined by the required data durability and the frequency of machine failures. Then adaptive recovery attempts to meet the deadline with minimal resource while allocating all remaining resource to executing new requests.

In order for this approach to work, first, we need a dynamic and fine-grained mechanism to control the resource dedicated to recovery. For this purpose, we split the whole recovery into multiple recovery requests, each fetching a subset of the state, and introduce a parameter I_{rec}: the system will execute I_{rec} client requests after it executes a recovery request. If no client requests arrive for a certain amount of time, however, this constraint can be relaxed. This parameter allows our approach to dynamically control the speed of recovery.

In order to meet the deadline, our approach tracks the progress of recovery: during recovery, a backup node needs to fetch state from an active node. The active node knows the total size of the state to be transferred and keeps track of how much data has already been transferred. It periodically checks the progress of recovery by comparing $\frac{fetched\ data}{total\ data}$ and $\frac{elapsed\ time}{deadline}$. If the former is smaller (larger) than the latter, it increases (decreases) recovery speed by decreasing (increasing) I_{rec}.

Agreement node recovery. A backup agreement node needs to fetch missing log entries from the leader. In this case, the leader will perform the above tracking and adaptive control mechanism.

Execution node recovery. A backup execution node needs to fetch both the latest snapshot from an active execution node and the log entries afterwards from the leader. In this case, both the active execution node and the leader need to perform the above tracking and adaptive control mechanism. These two procedures run in parallel and may compete for resource, but since our adaptive approach tries to meet the deadline with minimal resource and gives up excessive resource automatically, these procedures will eventually get a necessary share of the resource, as long as the system has enough resource for both of them.

Soft deadline. Adaptive recovery makes best effort to meet the deadline but cannot provide any guarantees for several reasons: first, if the deadline is too close, the system may not be able to meet the deadline even if it allocates all resources to recovery. Second, the adaptive approach takes time to monitor progress and adjust recovery speed, so if the recovery time is too short, our approach may not be effective. Finally, our approach relies on the assumption that the system throughputs when processing client requests and recovery requests are reasonably stable. If they can change rapidly, because of either hardware issues (e.g. contention on network) or software issues (e.g. some requests take much longer than others to process), our mechanism may not be accurate.

5 ThriftyPaxos

To demonstrate the effectiveness of our approach, we apply it to Paxos, a popular replication protocol, to build ThriftyPaxos. In this section, we present the detailed protocol of ThriftyPaxos. As one can imagine, it bears a significant resemblance to the standard Paxos protocol. For completeness, we present the whole ThriftyPaxos protocol, but we highlight the different part.

5.1 Overview

Paxos incorporates $2f+1$ replicas. Its key idea is that when a request is agreed by $f+1$ replicas as the next request, the request becomes *stable*, which means the decision will not be changed by future failures. Then all execution nodes execute the stable request.

A ThriftyPaxos service follows the same idea. It logically separates replicas into $2f+1$ agreement nodes and execution nodes (a pair of agreement and execution nodes can be collocated on the same machine or even in a single process). By following the calculations in Section 3, ThriftyPaxos activates $f+1$ of them and reserves f of them as backup (standard Paxos activates all of them).

To order client requests, ThriftyPaxos tries to assign a unique *slot number* to each client request: a client request with a lower slot number is ordered before one with a higher slot number. The safety property requires that each slot can be assigned to at most one client request.

In most of the time, the system elects one agreement node as the *leader*, which proposes the next request to execute. When failures or asynchronous events happen, new leaders may be elected, even if the old leader is still alive. Different leaders may make different proposals for the same slot, and to distinguish them, each leader is assigned a unique *epoch number* and each leader attaches this epoch number to each proposal it makes. A later elected leader has a higher epoch number than an early leader.

5.2 Basic protocol

(1) A client sends a request to the leader.

A client can have multiple outstanding requests and thus the system needs a unique identifier for each request to match a reply to a request. A classic approach to generate request ID is to combine client ID with the number of requests the client has already sent.

If the client does not get a reply in time, either because some messages are lost or because some replicas fail, it resends those request to all replicas.

(2) The leader checks the request and if the check passes, the leader proposes it as the next request.

The leader needs to check whether this is the expected request from the corresponding client. In order to check that, the leader maintains the last request ID it has proposed for each client. There are four possible cases:

(2.1) If $requestID = lastRequestID[client]+1$, then this is the appropriate request to propose. The leader sends a proposal $< epoch, slot, request >$ to $f+1$ agreement nodes including the leader itself (standard Paxos sends to $2f+1$ agreement nodes). The leader then needs to update corresponding state, such as *slot*.

(2.2) If $requestID > lastRequestID[client]+1$, it means some previous client's requests are lost. In this case, the leader can either cache the request or discard it if the cache is full. In either case, the leader needs to wait for the client to resend the missing requests.

(2.3) If $requestID < lastRequestID[client]+1$ and the request is in the leader's pending proposals, it means that the leader is processing the request. In this case, the leader performs no action.

(2.4) If $requestID < lastRequestID[client]+1$ and the request is not in the leader's pending proposals, it means the leader has already finished proposing this request. This can happen if some messages were lost so that the client did not receive the reply. In this case, the leader forwards the request to execution nodes (see Step (6.2)).

(3) When an agreement node receives a proposal, it decides whether to accept the proposal.

(3.1) If $proposal.epoch < this.epoch$ or if the agreement node is not active, the agreement node discards the proposal and notifies the leader.

(3.2) Otherwise, the agreement node accepts the proposal by logging it to disk and sending an acknowledgement to the leader. If $proposal.epoch > this.epoch$, the agreement node also updates its own epoch. Note that the agreement node can accept proposals out of order.

(4) The leader waits for $f+1$ acknowledgements.

If any agreement node responds that it has accepted a proposal with a higher epoch number, this leader gives up and will not process any further requests from the clients.

If any agreement node times out, the leader picks up one backup agreement node and sends an *ACTIVATE*

command to it (standard Paxos, of course, does not need this command because all replicas are active). Then the leader resends all pending proposals.

(5) The leader sends the agreed request to all active execution nodes.

If there are fewer than $f+1$ active execution nodes, the leader will send an *ACTIVATE* command to some backup execution nodes (standard Paxos does not need this).

(6) An execution node executes the request.

An execution node needs to execute requests in order. In order to achieve that, it maintains the last slot that it has already executed. When receiving a request, it first checks the slot number of the request.

(6.1) If $request.slot = lastslot + 1$, the request is the next to execute. The execution node executes the request and sends the reply to the client. It then checks whether there are any following requests in the cache ((6.3)) that can be executed.

(6.2) If $request.slot <= lastslot$, then this request has already been executed. This might happen if the reply message got lost and the client resent the request. In this case, the execution node should resend the reply to the client. To achieve that, the execution node needs to maintain a reply cache for replies it has already sent. To garbage collect replies, a client can piggyback in each of its request the ID of the latest received reply.

(6.3) If $request.slot > lastslot + 1$, the execution node caches the request or stores the request to disk if the cache is full. This could happen if a previous message was lost or the execution node is in recovery. For the former case, the execution node asks the leader to resend missing requests. For the later case, the execution node has to wait for the recovery to complete.

To garbage collect logs on the agreement nodes ((3.2)), ThriftyPaxos requires execution nodes to periodically take snapshots of their states, promising that they will not need earlier requests in the future. ThriftyPaxos performs garbage collection when $f+1$ execution nodes complete taking snapshots (standard Paxos asks an agreement node to garbage collect its log when its corresponding execution node takes a snapshot. This is not different from ThriftyPaxos when $f+1$ execution nodes are active, but since in ThriftyPaxos, sometimes there could be different number of agreement nodes and execution nodes, we use a more explicit rule).

5.3 Failure recovery

Recovering execution nodes. When an execution node is not responding, the leader sends an *ACTIVATE* command to a backup execution node, asking it to rebuild its state. Meanwhile, ThriftyPaxos proceeds with the remaining active execution nodes. To rebuild state, the backup execution node may need to fetch the latest snapshot from an active execution node and fetch following logs from the leader. Since the system is processing clients' requests during recovery, and the backup node cannot process them at the moment, it will cache these requests first in memory and then in a log file on disk if the memory buffer is full. After state transfer is complete, the backup node needs to load the snapshot and replay all logs afterwards.

We have applied adaptive recovery at the leader and at the active execution node to control the recovery speed of the backup execution node. When trying to meet the deadline, adaptive recovery considers recovery as complete when state transfer is done, because at that moment, the backup node has the same state as an active one, which means data durability is restored and the system can garbage collect logs on agreement nodes. Therefore, later operations, including loading snapshot and replaying logs, are not considered as part of recovery.

Recovering agreement nodes. When a non-leader agreement node is not responding, the leader will send an *ACTIVATE* command to a backup agreement node, asking it to join the agreement protocol immediately. Meanwhile, to restore data durability, the backup node fetches missing logs from the leader in the background. We have applied adaptive recovery at the leader to control the speed of fetching missing logs.

When the leader is not responding, a new leader is elected. The new leader collects logs from other agreement nodes and re-proposes pending requests. ThriftyPaxos uses the same protocol as standard Paxos for leader switch (backup agreement nodes also need to participate in a leader switch). This part is out of the control of adaptive recovery, because the system is blocked anyway. The developer can make a trade-off between performance and availability by setting the snapshot interval, since only requests after the last snapshot need to be re-proposed during a leader switch. Note that this trade-off exists in both ThriftyPaxos and standard Paxos.

6 Implementation

We have implemented ThriftyPaxos from scratch in Java. This section discusses some implementation details.

Failure detection. A highly available system must be able to detect failures quickly. Like standard Paxos, ThriftyPaxos does not rely on the accuracy of failure detection for safety and thus can use aggressive approaches (e.g. short timeout) for better availability. Our experiments, however, show that if timeout interval is too short, performance can be unstable, because even a minor abnormal event, such as a long disk write, can trigger an unnecessary recovery. Our implementation incorporates ideas from accurate failure detection techniques [42]:

when failures can be detected for certain, our system should take actions immediately. To cover failures that cannot be accurately detected, we use a timeout of five seconds. Our current implementation only incorporates part of the functionalities of those accurate failure detection techniques, and we leave the full incorporation as future work.

Out-of-order logging. Agreement nodes may need to log requests out of order on disks. To achieve efficient logging, ThriftyPaxos incorporates a design that is similar to SSTable in Bigtable [17]: if the slot number of the new request is larger than the last one in the current file, ThriftyPaxos appends it to the current file; otherwise, ThriftyPaxos closes the current file, creates a new one, and appends the next log entry into the new file. Such design guarantees that each log file is sequential, and thus it is simple to perform operations like log scan (merge sort) and garbage collection. Furthermore, when there are no out-of-order requests, logging is fully sequential and thus can fully utilize a hard drive. Note that this design may not be suitable for a BFT protocol, in which a faulty primary can attack the system by sending out-of-order requests deliberately and creating a lot of small files: storing logs in random-accessible stores like Berkeley DB [7] may be more appropriate, although their overhead is usually higher.

Leader election. During leader switch, the new leader needs to collect logs from other agreement nodes and repropose pending requests. To reduce I/O traffic during leader election, ThriftyPaxos gives a preference to an active agreement node as the new leader, because it has more logs compared to a backup node.

When to start recovery. Previous study shows that it may be a waste of resource to start recovering a node right after it fails, because a crashed node is likely to come back soon. In practice, Google starts to recover a node if it cannot come back in 15 minutes [25]. Our approach follows the same idea: when a backup node fails, ThriftyPaxos waits a while to see whether it can come back before triggering recovery. We use a relatively short interval (5 minutes) to shorten the length of experiments.

7 Evaluation

Our evaluation tries to answer three questions:

- What is the performance of ThriftyPaxos, when there are no failures?
- What is the availability of ThriftyPaxos, when failures occur?
- Can adaptive recovery meet the deadline under different settings?

To address these questions, we have applied ThriftyPaxos to replicate H2 [27], a database system, and RemoteHashMap, a benchmark application built by us.

H2. H2 is a light-weight database system implemented in Java. It is often used as an embedded data store for larger projects, such as Hadoop [28]. To apply ThriftyPaxos to H2, we have modified H2 to send and receive messages through ThriftyPaxos. We configure H2 to store all tables in memory while keeping logs and snapshots on disks. To achieve quick snapshot, we configure H2 to store data in Btrfs [12], a Linux file system that supports copy on write (COW) snapshot: whenever an execution node needs to perform a snapshot, it asks Btrfs to perform a snapshot, which usually takes only several seconds. We ran TPC-C [55] over H2 to measure its performance. To avoid non-determinism, we ran H2 in single-threaded mode, which of course limits its performance. Efficient deterministic multithreading is an orthogonal topic that has been studied in many other works [2, 8, 9, 21, 22, 43] and we leave the incorporation as future work.

RemoteHashMap. To test ThriftyPaxos under various workloads, we have built RemoteHashMap, an application that allows clients to get and set key value pairs on a remote server. RemoteHashMap allows us to change system parameters (e.g. request size, snapshot size) arbitrarily to test different aspects of ThriftyPaxos.

We run all H2 experiments on three Dell R730 machines, which are equipped with 16 cores, 64GB of memory, one SSD drive, and seven hard drivers. We store H2's data on the SSD drive and store agreement nodes' data on one hard drive. We run all RemoteHashMap experiments on 7 Dell R220 machines, which are equipped with 8 cores, 16GB of memory, and two hard drivers. We store RemoteHashMap's data on one hard drive and store agreement nodes' data on the other hard drive.

We compare ThriftyPaxos to standard Paxos and Cheap Paxos. We implement standard Paxos and Cheap Paxos by slightly modifying ThriftyPaxos: standard Paxos chooses all $2f + 1$ replicas as active ones; Cheap Paxos sets I_{rec} to 0 during recovery, indicating it cannot execute client requests until recovery completes. In all experiments, we collocate an agreement node and an execution node on a same machine.

7.1 Performance

Our first set of experiments aims at comparing the performance of ThriftyPaxos to that of standard Paxos, when there are no failures. Cheap Paxos' protocol in the failure-free case is exactly the same as ThriftyPaxos, so we do not include it in the comparison.

As shown in Figure 1, when running TPC-C over H2, the performance of ThriftyPaxos and standard Paxos are almost identical. This is because the bottleneck of the system lies in the execution of requests as a result of

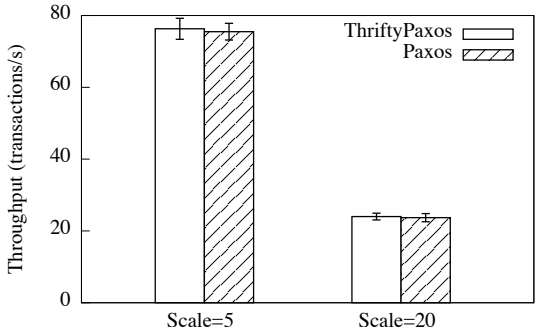


Figure 1: Throughput of TPC-C over replicated H2

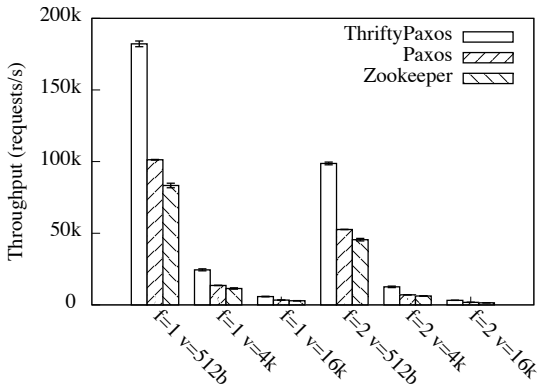


Figure 2: Throughput of writing to replicated RemoteHashMap (v is value size). We show ZooKeeper as a comparison.

single-threaded execution: our profiling shows that one CPU core is fully utilized.

Figure 2 shows the throughput of writing key-value pairs to RemoteHashMap, whose execution is relatively light and agreement is the bottleneck. In this case, ThriftyPaxos outperforms standard Paxos by 73% to 88%, because in ThriftyPaxos, the leader, which is the bottleneck, only needs to send messages to f replicas, while in standard Paxos, the leader needs to send to $2f$. We also show the throughput of ZooKeeper [29], a mature open-source software whose update protocol is similar to standard Paxos. Since we are interested in agreement throughput, we disable the "sync" call when logging to disk for all systems. ZooKeeper cannot serve as a direct comparison to ThriftyPaxos, because they have different features and optimizations. We show ZooKeeper's throughput only to present that our implementation provides a reasonable throughput.

Figure 3 compares the latencies of ThriftyPaxos and standard Paxos by writing small key-value pairs to RemoteHashMap. As shown in the figure, when throughput is low, the latency of ThriftyPaxos is almost identical to that of standard Paxos: this is because they need the same rounds of message exchanges to execute a request. Latency of standard Paxos jumps at the throughput of around 100k requests per second, because standard Paxos has reached its maximum throughput, while for ThriftyPaxos, such jump occurs at around 190k requests per second, because ThriftyPaxos can provide a higher throughput.

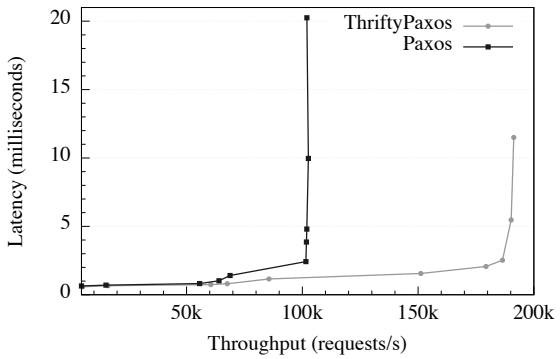


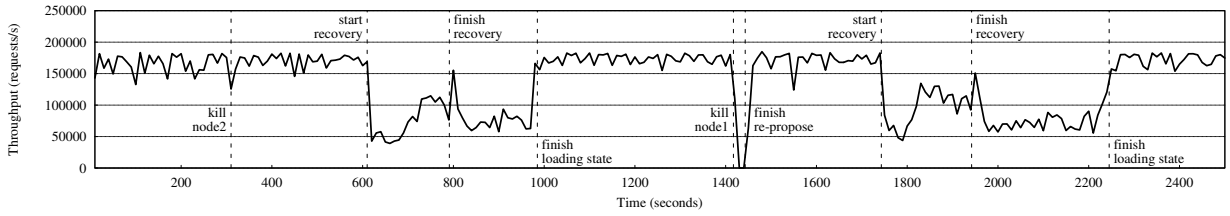
Figure 3: Latencies of ThriftyPaxos and standard Paxos (RemoteHashMap with f=1 and v=512b)

7.2 Availability

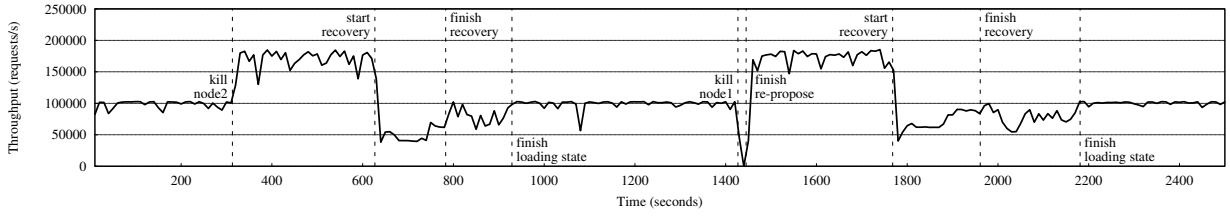
Our second set of experiments compare the availability of ThriftyPaxos to that of standard Paxos and Cheap Paxos, when failures occur.

We use RemoteHashMap in this set of experiments, because RemoteHashMap's much higher throughput creates a higher pressure on network and disk I/Os, incurring more contention against recovery. To measure availability, we kill a non-leader replica at 300 seconds, and kill the leader at 1400 seconds, for all three systems. Since an agreement node and an execution node are collocated on the same machine, both of them will be killed.

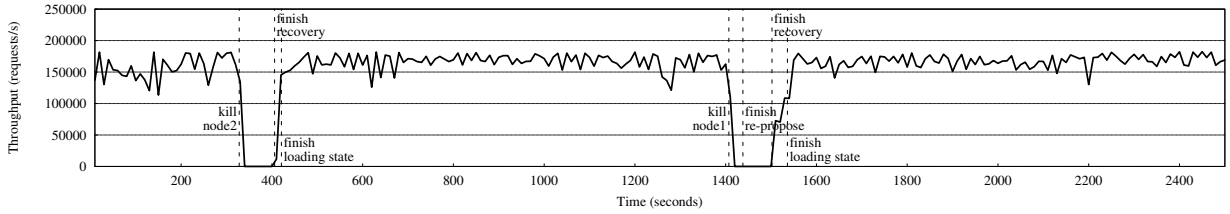
As shown in Figure 4, the behavior of ThriftyPaxos and standard Paxos are quite similar: when a non-leader replica fails, the system can continue processing requests; when the leader fails, the system needs to elect a new leader, which may block the system for a short while. In either case, since the failed replica does not come back in five minutes, the system needs to rebuild its state on another replica: rebuilding incurs network and disk I/Os, degrading system performance. The behaviors of ThriftyPaxos and standard Paxos are different in two ways: first, after a replica fails in standard Paxos, system throughput jumps. This is because the leader, which is the bottleneck in standard Paxos, needs to send fewer messages after a failure. ThriftyPaxos, on the other hand, has already exploited this opportunity in the failure-free case and thus its throughput remains stable after a failure. Second, ThriftyPaxos' loading time after recovery is slightly longer than that of standard Paxos. This is because ThriftyPaxos' throughput during



(a) ThriftyPaxos

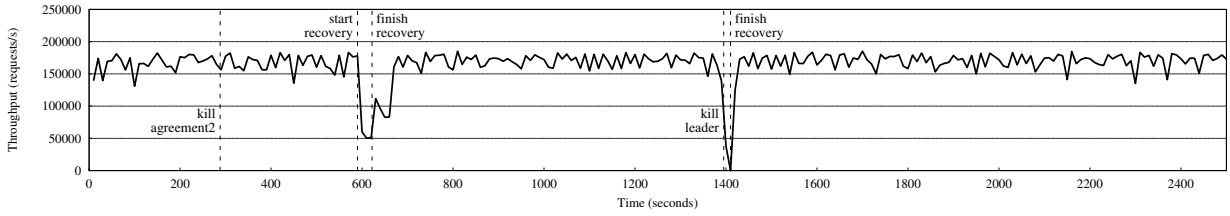


(b) Standard Paxos

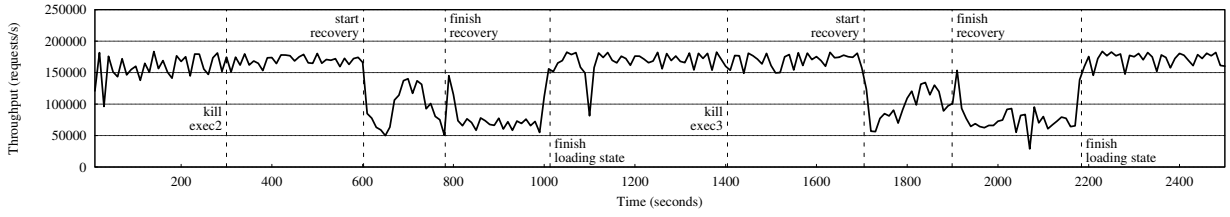


(c) Cheap Paxos

Figure 4: Availability of ThriftyPaxos, standard Paxos, and Cheap Paxos (f=1; node1 is the leader; v=512b).



(a) Impact of recovering agreement node



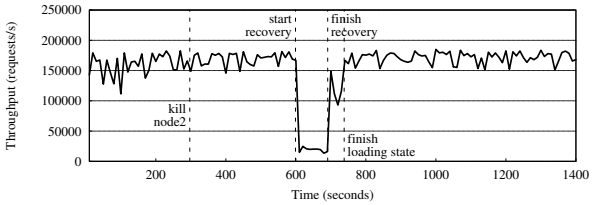
(b) Impact of recovering execution node

Figure 5: Impact of recovering agreement node and recovering execution node

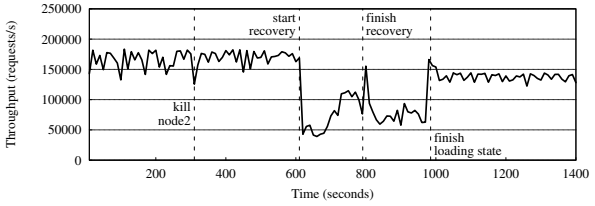
recovery is higher than that of standard Paxos (because, once again, ThriftyPaxos sends fewer messages for each request), and thus ThriftyPaxos needs to replay more log entries afterwards.

Cheap Paxos shows a different behavior: when either replica fails, Cheap Paxos needs to rebuild its state before it can process new requests. Therefore, the system is halted during node recovery. While its recovery takes 78-96 seconds in our experiments to transfer about 8GBs of state (including both snapshot and following logs), it may halt longer when application's state is larger.

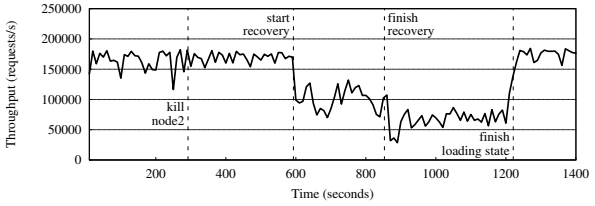
We further decouple recovery of agreement node and recovery of execution node to understand their individual impact. As shown in Figure 5, recovery of execution node certainly has a bigger impact on performance, because it needs to transfer more state. Recovery of agree-



(a) Deadline=100 seconds

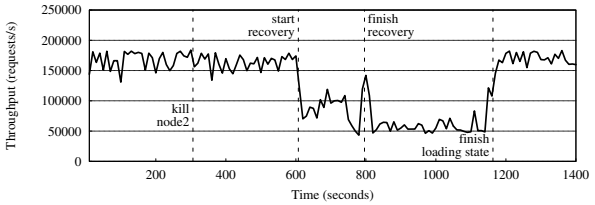


(b) Deadline=200 seconds

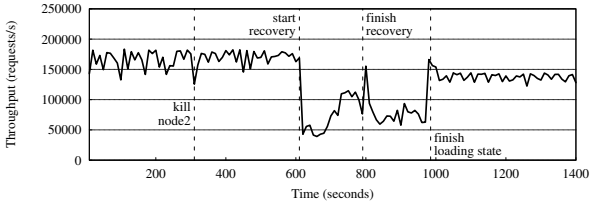


(c) Deadline=300 seconds

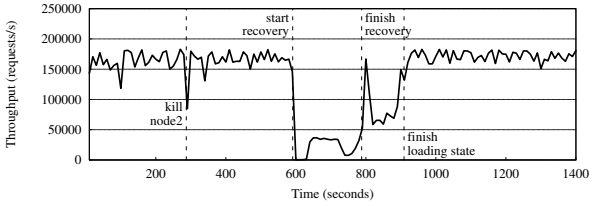
Figure 6: Recovery with different deadlines (snapshot size=5G; node2 is a non-leader replica)



(a) Snapshot size=1G



(b) Snapshot size=5G



(c) Snapshot size=15G

Figure 7: Recovery with different snapshot sizes (deadline=200 seconds; node2 is a non-leader replica)

ment node is lighter in general, but the failure of the leader can halt the system for a short while.

7.3 Adaptive recovery

Our last set of experiments measures whether adaptive recovery can meet the deadline. We set different deadlines and sizes of snapshots for this set of experiments.

Figure 6 shows the impact of different recovery deadlines. As one can observe, closer deadline, which means the system must dedicate more resource to recovery, can cause a sharper degradation of performance during recovery: we tried deadlines 100, 200, and 300, and the average throughputs during recovery are 22240, 74690, and 99339 respectively.

Figure 7 shows the impact of different snapshot sizes. As one can observe, bigger snapshot, which means the system must dedicate more resource to recovery, can cause a sharper degradation of performance during recovery. Once again, our adaptive recovery mechanism successfully meets the deadline in all experiments.

In both figures, a higher throughput during recovery increases the time to load state afterwards. This is simply because higher throughput during recovery increases the number of log entries to be replayed after recovery. Faster disks or a larger memory buffer may reduce such replaying time.

Our adaptive recovery mechanism successfully meets the deadline in all experiments. Although these results may not be extended to other applications, as discussed in Section 4, they demonstrate the efficacy of adaptive recovery for applications that can provide stable throughputs when processing client requests and recovery requests. Furthermore, in all experiments, the recovery completion time is close to the deadline. This demonstrates that adaptive recovery achieves its goal: to meet the deadline with minimal resource while using all remaining resource to process new requests.

8 Related work

While Section 2 already presents important related works in detail, this section presents a broader survey.

State machine replication. State machine replication (SMR) [52] replicates an application's state machine on multiple machines to tolerate failures. Based on the types of failures it can tolerate, SMR protocols can be broadly classified into two categories: the Paxos family [33, 34, 48] that can tolerate machine crashes and timing errors, and the Byzantine Fault Tolerance (BFT) family [1, 6, 15, 18, 20, 30, 38] that can tolerate arbitrary errors.

Separating agreement from execution. Paxos separates agreement (acceptors) from execution (learners) to clarify its protocol [16, 33, 34, 39, 49, 56]. Yin et al. [60]

observe that for BFT protocols, the number of required execution replicas could be fewer than that of the agreement nodes. This observation is inherited in later works, such as UpRight [18], Zyzzyva [30], and ZZ [59].

On-demand instantiation. Cheap Paxos [37] and ZZ [37] activate minimal number of replicas first and activate backup ones when active ones are unresponsive. However, they require a backup node to rebuild its state before it can process requests, suffering from the availability problem. Distler et al. [24] proposes to split application state into multiple objects and perform on-demand instantiation for each object: this approach can alleviate the availability problem, but since it needs to maintain a log for each object, the space overhead is significantly magnified. Parallel recovery techniques [17, 50] can effectively reduce recovery time and thus can alleviate the availability problem of on-demand instantiation, but recovering a node right after it fails is a waste of resource, since most crashed nodes can come back soon [25].

Lazy recovery. Lazy recovery [31, 32] is widely used in many systems. For example, Google File System [26] starts recovering a failed node if it cannot come back in 15 minutes. Silberstein et al. [54] applies lazy recovery to erasure-coding systems to reduce recovery overhead.

Other optimizations. Vertical Paxos [36] proposes to strengthen primary backup protocols with the help of a centralized Paxos service. Our approach provides a more general approach that does not require an additional Paxos service and that is applicable to BFT protocols.

Falcon [42] and its following works [40, 41] attempt to build accurate failure detectors, which make Paxos unnecessary. However, they rely on routers to monitor the status of machines and thus can become unavailable when routers fail, which can happen in today's datacenters [25]. Furthermore, it is inapplicable to BFT systems.

Gaios [10] and ZooKeeper [29] execute read requests on only one replica. Moraru et al. propose a quorum lease mechanism to improve the throughput and reduce the latency of read operations in Paxos [46]. Fast Paxos [35], Speculative Paxos [51], and Zyzzyva [30] introduce a fast path to reduce latencies of all requests. Mencius [44] and EPaxos [45] allow multiple leaders to propose requests in parallel to achieve load balancing. These optimizations are orthogonal to our approach. While Section 3 has discussed the feasibility of combining those fast-path optimizations with our approach, it would be interesting to investigate whether it is possible to combine other optimizations with our approach.

9 Conclusion

Whether to pay the cost of a strong replication protocol has been a painful question for developers. Instead of inventing new protocols, this paper presents a general approach to reason about the necessary conditions for correctness and availability in existing protocols. It shows that, with a deeper understanding of existing protocols, we can reduce their replication cost while maintaining their correctness and availability properties.

Acknowledgements

Many thanks to our shepherd Mahesh Balakrishnan and to the anonymous reviewers for their insightful comments. Lorenzo Alvisi and Manos Kapritsos provided invaluable feedbacks on early versions of this project. This material is based in part upon work supported by the National Science Foundation under Grant Number CNS-1566403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-Scalable Byzantine Fault-Tolerant Services. In *SOSP*, 2005.

[2] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *OSDI*, 2010.

[3] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, 2011.

[4] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. NSDI'12, 2012.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. SOSP '13.

[6] Rida A. Bazzi. Synchronous Byzantine quorum systems. *Distributed Computing*, 13(1):45–52, 2000.

[7] Berkeley DB. `http://www.oracle.com/technetwork/database/database/technologies/berkeleydb/downloads/index.html`.

[8] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *SIGARCH Comput. Archit. News*, 2010.

[9] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dOS. In *OSDI*, 2010.

[10] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines as the Basis of a High-Performance Data Store. In *NSDI*, 2011.

[11] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based Fault Tolerance. *ACM Transanctions on Computer Systems*, 14(1):80–107, February 1996.

[12] Btrfs. `https://btrfs.wiki.kernel.org/index.php/Main_Page`.

[13] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Primary-Backup Protocols: Lower Bounds and Optimal Implementations. In *CDCCA*, 1992.

[14] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[15] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transanctions on Computer Systems*, 20(4):398–461, November 2002.

[16] T. Chandra, R. Griesmer, and J. Redstone. Paxos made live – an engineering perspective. In *Proc. 26th PODC*, June 2007.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI*, 2006.

[18] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. UpRight Cluster Services. In *SOSP*, 2009.

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[20] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *OSDI*, 2006.

[21] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos Made Transparent. In *SOSP*, 2015.

[22] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *SOSP*, 2011.

[23] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High Availability via Asynchronous Virtual Machine Replication. In *NSDI*, 2008.

[24] Tobias Distler and Rüdiger Kapitza. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In *Eurosys*, 2011.

[25] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP*, 2003.

[27] H2. The H2 home page. http://www.h2database.com.

[28] Hadoop. `http://hadoop.apache.org/core/`.

[29] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX ATC*, 2010.

[30] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *SOSP*, 2007.

[31] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, 1992.

[32] Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: exploiting the semantics of distributed services. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 43–57, New York, NY, USA, 1990. ACM.

[33] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[34] Leslie Lamport. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, December 2001.

[35] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[36] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, 2009.

[37] Leslie Lamport and Mike Massa. Cheap Paxos. In *DSN*, 2004.

[38] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[39] Butler Lampson. The abcds of paxos. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, page 13, New York, NY, USA, 2001. ACM.

[40] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, 2013. USENIX.

[41] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 9:1–9:16, New York, NY, USA, 2015. ACM.

[42] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *SOSP*, 2011.

[43] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *SOSP*, 2011.

[44] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *OSDI*, 2008.

[45] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[46] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *Proc. 5th ACM Symposium on Cloud Computing (SOCC)*, Seattle, WA, November 2014.

[47] MySQL. `http://www.mysql.com`.

[48] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proc. 7th PODC*, 1988.

[49] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[50] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *SOSP*, 2011.

[51] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 43–57, Oakland, CA, May 2015. USENIX Association.

[52] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[53] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST*, 2010.

[54] Mark Silberstein, Lakshmi Ganesh, Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 15:1–15:7, New York, NY, USA, 2014. ACM.

[55] Transaction Processing Performance Council. The TPC-C home page. http://www.tpc.org/tpcc/.

[56] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, February 2015.

[57] Robbert van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, 2004.

[58] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating Data and Metadata for Efficient and Available Storage Replication. In *USENIX ATC*, 2012.

[59] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the Art of Practical BFT. In *Eurosys*, 2011.

[60] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *SOSP*, 2003.

Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing

Alex D. Breslow* Dong Ping Zhang Joseph L. Greathouse Nuwan Jayasena Dean M. Tullsen
AMD Research *AMD Research* *AMD Research* *AMD Research* *UC San Diego*

Abstract

Hash tables are important data structures that lie at the heart of important applications such as key-value stores and relational databases. Typically bucketized cuckoo hash tables (BCHTs) are used because they provide high-throughput lookups and load factors that exceed 95%. Unfortunately, this performance comes at the cost of reduced memory access efficiency. Positive lookups (key is in the table) and negative lookups (where it is not) on average access 1.5 and 2.0 buckets, respectively, which results in 50 to 100% more table-containing cache lines to be accessed than should be minimally necessary.

To reduce these surplus accesses, this paper presents the Horton table, a revamped BCHT that reduces the expected cost of positive and negative lookups to fewer than 1.18 and 1.06 buckets, respectively, while still achieving load factors of 95%. The key innovation is remap entries, small in-bucket records that allow (1) more elements to be hashed using a single, primary hash function, (2) items that overflow buckets to be tracked and rehashed with one of many alternate functions while maintaining a worst-case lookup cost of 2 buckets, and (3) shortening the vast majority of negative searches to 1 bucket access. With these advancements, Horton tables outperform BCHTs by 17% to 89%.

1 Introduction

Hash tables are fundamental data structures that are ubiquitous in high-performance and big-data applications such as in-memory relational databases [12, 17, 40] and key-value stores [20, 24]. Typically these workloads are read-heavy [4, 62]: the hash table is built once and is seldom modified in comparison to total accesses. A hash table that is particularly suited to this behavior is a bucketized cuckoo hash table (BCHT), a type of open-addressed hash table.[1] BCHTs group their cells into buckets: associative blocks of two to eight slots, with each slot capable of storing a single element.

When inserting an element, BCHTs typically select between one of two independent hash functions, each of which maps the key-value pair, call it KV, to a different *candidate bucket*. If one candidate has a free slot, KV is inserted. In the case where neither has spare slots, BCHTs resort to *cuckoo hashing*, a technique that resolves collisions by evicting and rehashing elements when too many elements vie for the same bucket. In this case, to make room for KV, the algorithm selects an element, call it KV', from one of KV's candidate buckets, and replaces it with KV. KV' is then subsequently rehashed to its alternate candidate using the remaining hash function. If the alternate bucket for KV' is full, KV' evicts yet another element and the process repeats until the final displaced element is relocated to a free slot.

Although these relocations may appear to incur large performance overheads, prior work demonstrates that most elements are inserted without displacing others and, accordingly, that BCHTs trade only a modest increase in average insertion and deletion time in exchange for high-throughput lookups and load factors that often exceed 95% with greater than 99% probability [19], a vast improvement over the majority of other techniques [60, 64].

BCHTs, due to this space efficiency and unparalleled throughput, have enabled recent performance breakthroughs in relational databases and key-value stores on server processors [20, 60, 64] as well as on accelerators such as GPUs [71], the Xeon Phi [15, 60], and the Cell processor [34, 64]. However, although BCHTs are higher-performing than other open addressing schemes [60, 64], we find that as the performance of modern computing systems becomes increasingly constrained by memory bandwidth [1, 11, 52, 70], they too suffer from a number of inefficiencies that originate from how data is fetched when satisfying queries.

Carefully coordinating table accesses is integral to throughput in hash tables. Because of the inherent randomness of hashing, accesses to hash tables often exhibit poor temporal and spatial locality, a property that causes hardware caches to become increasingly less effective as tables scale in size. For large tables, cache lines containing previously accessed hash table buckets are frequently evicted due to capacity misses before they are touched again, degrading performance and causing applications to become memory-bandwidth-bound once the table's working set cannot be cached on-chip. Given these concerns, techniques that reduce accesses to additional cache lines in the table prove invaluable when optimizing hash table performance and motivate the need to identify and address the data movement inefficiencies that are prevalent in BCHTs.

Consider a BCHT that uses two independent hash functions to map each element to one of two candidate buckets. To load balance buckets and attain high load factors, recent work on BCHT-based key-value stores inserts each element into the candidate bucket with the least load [20, 71], which means that we expect half of the elements to be hashed by each function. Consequently, on positive lookups, where the queried key is in the table, 1.5 buckets are expected to be examined. Half of

*This author is also a PhD student at UC San Diego. Send correspondence regarding the work to abreslow@cs.ucsd.edu.

[1] Under open addressing, an element may be placed in more than one location in the table. Collisions are resolved by relocating elements within the table rather than spilling to table-external storage.

the items can be retrieved by examining a single bucket, and the other half require accessing both. For negative lookups, where the queried key is not in the table, 2 lookups are necessary. Given that the minimum number of buckets that might need to be searched (for both positive and negative lookups) is 1, this leaves a very significant opportunity for improvement.

To this end, this paper presents Horton tables,[2] a carefully retrofitted bucketized cuckoo hash table, which trades a small amount of space for achieving positive and negative lookups that touch close to 1 bucket apiece. Our scheme introduces **remap entries**, small and succinct in-bucket records that enable (1) the tracking of past hashing decisions, (2) the use of many hash functions for little to no cost, and (3) most lookups, negative and positive alike, to be satisfied with a single bucket access and at most 2. Instead of giving equal weight to each hash function, which leads to frequent fetching of unnecessary buckets, we employ a single **primary hash function** that is used for the vast majority of elements in the table. By inducing such heavy skew, most lookups can be satisfied by accessing only a single bucket. To permit this biasing, we use several secondary hash functions (7 in our evaluations) to rehash elements to alternate buckets when their preferred bucket lacks sufficient capacity to directly store them. Rather than forgetting our choice of secondary hash function for remapped elements, we convert one of the slots in each bucket that overflows into a **remap entry array** that encodes which hash function was used to remap each of the overflow items. It is this ability to track all remapped items at low cost, both in terms of time and space, that permits the optimizations that give Horton tables their performance edge over the prior state-of-the-art.

To achieve this low cost, instead of storing an explicit tag or fingerprint (a succinct hash representation of the remapped object) as is done in cuckoo filters [21] and other work [8, 13], we instead employ implicit tags, where the index into the remap entry array is a tag computed by a hash function H_{tag} on the key. This space optimization permits all buckets to use at most 64 bits of remap entries in our implementation while recording all remappings, even for high load factors and tables with billions of elements. As a further optimization, we only convert the last slot of each bucket into a remap entry array when necessary. For buckets that do not overflow, they remain as standard buckets with full capacity, which permits load factors that exceed 90 and 95 percent for 4- and 8-slot buckets, respectively.

Our main contributions are as follows:

- We develop and evaluate Horton tables and demonstrate speed improvements of 17 to 89% on graphics processors (GPUs). Although our algorithm is not specific to GPUs, GPUs represent the most efficient platform for the current state of the art, and thus it is important to demonstrate the effectiveness of Horton tables on the same platform.
- We present algorithms for insertions, deletions, and lookups on Horton tables.
- We conduct a detailed analysis of Horton tables by deriving and empirically validating models for their expected data movement and storage overheads.
- We investigate additional optimizations for insertions and deletions that further reduce data movement when using multiple hash functions, reclaiming remap entries once their remapped elements are deleted, even when they are shared by two or more table elements.

This paper is organized as follows: In Section 2 we elaborate on the interplay between BCHTs and single instruction multiple data (SIMD) processors, in Section 3 we describe BCHTs, in Section 4 we provide a high-level overview of Horton tables, in Section 5 we describe the lookup, insertion, and deletion algorithms for Horton tables, and then in Section 6 we present our models for Horton tables that include the cost of insertions, deletions, and remap entry storage. Section 7 covers our experimental methodology, and Section 8 contains our performance and model validation results. Related work is described in Section 9 and Section 10 concludes.

2 The Role of SIMD

In key places in this paper, we make references to SIMD and GPU architectures. Although not necessary for understanding our innovations, these references are present due to SIMD's importance for high-throughput implementations of in-memory hash tables and BCHTs in particular.

Recent work in high-performance databases that leverages BCHTs has shown that SIMD implementations of BCHTs, as well as larger data processing primitives, are often more than $3\times$ faster than the highest performing implementations that use scalar instructions alone [6, 44, 60]. These SIMD implementations enable billions of lookups per second to be satisfied on a single server [60, 71], an unimaginable feat only a few years ago. At the same time, SIMD implementations of BCHTs are faster than hash tables that use other open addressing methods [60, 64]. Such implementations are growing in importance because both CPUs and GPUs require writing SIMD implementations to maximize performance-per-watt and reduce total cost of ownership.

For these reasons, we focus on a SIMD implementation of BCHTs as a starting point and endeavor to show that all further optimizations provided by Horton tables not only have theoretical models that justify their performance edge (Section 6) but also that practical SIMD implementations deliver the performance benefits that the theory projects (Section 8).[3]

3 Background on BCHTs

In this section, we describe in detail BCHTs and the associated performance considerations that arise out of their interaction with the memory hierarchy of today's systems.

To begin, we illustrate two common scenarios that are triggered by the insertion of two different key-value pairs KV_1 and KV_2 into the hash table, as shown in Figure 1. Numbered rows correspond to buckets, and groups of

[2] Named for elephants' remarkable recall powers [18].

[3] For a primer on SIMD and GPGPU architectures, we recommend these excellent references: H&P (Ch. 4) [30] and Keckler et al. [39].

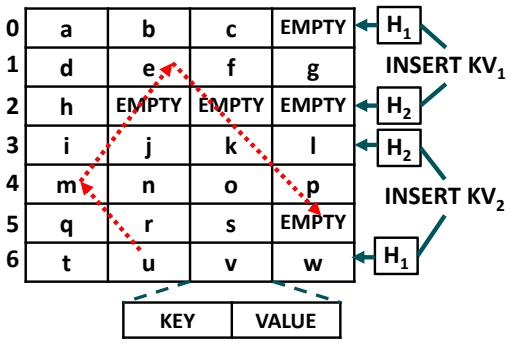


Figure 1: *Inserting items* $\mathbf{KV_1}$ *and* $\mathbf{KV_2}$ *into a BCHT*

four cells within a row to slots. In this example, H_1 and H_2 correspond to the two independent hash functions that are used to hash each item to two candidate buckets (0 and 2 for KV_1, 3 and 6 for KV_2). Both H_1 and H_2 are a viable choice for KV_1 because both buckets 0 and 2 have free slots. Deciding which to insert into is at the discretion of the algorithm (see Section 3.3 for more details).

For KV_2, both H_1 and H_2 hash it to buckets that are already full, which is resolved by evicting one of the elements (in this case u), and relocating it and other conflicting elements in succession using a different hash function until a free slot is found.[4] So e moves to the empty position in bucket 5, m to e's old position, u to m's old position, and KV_2 to u's old position. Li, et al. demonstrated that an efficient way to perform these displacements is to first conduct a breadth-first search starting from the candidate buckets and then begin moving elements only once a path to a free slot is discovered [47].

3.1 State-of-Practice Implementation

A number of important parameters affect the performance of BCHTs. In particular, the number of hash functions ($\boldsymbol{f}$) and the number of slots per bucket ($\boldsymbol{S}$) impact the achievable load factor (i.e., how full a table can be filled) as well as the expected lookup time. A hash table with more slots per bucket can more readily accommodate collisions without requiring a rehashing mechanism (such as cuckoo hashing) and can increase the table's load factor. Most implementations use four [20,60,64] or eight [71] slots per bucket, which typically leads to one to two buckets per hardware cache line. Using more slots comes at the cost of more key comparisons on lookups, since the requested element could be in any of the slots.

Increasing f, the number of hash functions, allows a key-value pair to be mapped to more buckets, as each hash function maps the item to one of f different buckets. This improved flexibility when placing an item permits the hash table to achieve a higher load factor. However, on lookups, more buckets need to be searched because the element could be in more locations. In practice, $f = 2$ is used most often because it permits sufficient flexibility in where keys are mapped without suffering from having to search too many buckets [20, 54, 63].

BCHTs are primarily designed for fast lookups. The get operation on any key requires examining the contents of at most f buckets. Because buckets have a fixed width, the lookup operation on a bucket can be unrolled and efficiently vectorized. These traits allow efficient SIMD implementations of BCHTs that achieve lookup rates superior to linear probing and double-hashing-based schemes on past and present server architectures [60, 64] and accelerators such as Intel's Xeon Phi [60].

[4] So in this example, elements on the chain that were originally hashed with H_1 would be rehashed using H_2 and vice versa.

3.2 Memory Traffic on Lookups

Like prior work, we divide lookups into two categories: (1) positive, where the lookup succeeds because the key is found in the hash table, and (2) negative where the lookup fails because the key is not in the table.

Prior work diverges on the precise method of accessing the hash table during lookups. The first method, which we term **latency-optimized**, always accesses all buckets where an item may be found [47, 64]. Another technique, which we call **bandwidth-optimized**, avoids fetching additional buckets where feasible [20, 71].

Given f independent hash functions where each of them maps each item to one of f candidate buckets, the latency-optimized approach always touches f buckets while the bandwidth-optimized one touches, on average, $(f+1)/2$ buckets on positive lookups and f buckets on negative lookups. For our work, we compare against the **bandwidth-optimized** approach, as it moves less data on average. Reducing such data movement is a greater performance concern on throughput-oriented architectures such as GPUs, since memory latency is often very effectively hidden on these devices [23]. Thus, we compare against the more bandwidth-oriented variant of BCHT, which searches 1.5 buckets instead of 2 (or more, if there are more hash functions) for positive lookups.

3.3 Insertion Policy and Lookups

Unlike the latency-optimized scheme, the bandwidth-optimized algorithm searches the buckets in some defined order. If an item is found before searching the last of the f candidate buckets, then we can reduce the lookup's data movement cost by skipping the search of the remaining candidate buckets. Thus if f is 2, and we call the first hash function H_1 and the second H_2, then the average lookup cost across all inserted keys is `1 * (fraction of keys that use` H_1`) + 2 * (fraction of keys that use` H_2`)`. Therefore, the insertion algorithm's policy on when to use H_1 or H_2 affects the lookup cost.

Given that hash tables almost always exhibit poor temporal and spatial locality, hash tables with working sets that are too large to fit in caches are bandwidth-bound and are quite sensitive to the comparatively limited off-chip bandwidth. In the ideal case, we therefore want to touch as few buckets as possible. If we can strongly favor using H_1 over H_2 during insertions, we can reduce the percentage of buckets that are fetched that do not contain the queried key, which reduces per-lookup bandwidth requirements as well as cache pollution, both of which improve lookup throughput.

Existing high-throughput, bandwidth-optimized BCHT implementations [20, 71] attempt to load-balance buckets on insertion by examining all buckets the key can map to and placing elements into the buckets with the most free slots. As an example, in Figure 1, KV_1 would be placed in the bucket hashed to by H_2. The

intuition behind this load balancing is that it both reduces the occurrence of cuckoo rehashing, which is commonly implemented with comparatively expensive atomic swap operations, and increases the anticipated load factor. Given this policy, H_1 and H_2 are both used with equal probability, which means that 1.5 buckets are searched on average for positive lookups. We refer to this approach as the **load-balancing baseline**.

An alternative approach is to insert items into the first candidate bucket that can house them. This technique reduces the positive lookup costs, since it favors the hash functions that are searched earlier. We refer to this as the **first-fit** heuristic. As an example, in Figure 1, KV_1 would be placed in the final slot of the top bucket of the table even though bucket 2 has more free slots. This policy means that items can be located with fewer memory accesses, on average, by avoiding fetching candidate buckets that follow a successful match. When the table is not particularly full, most elements can be inserted and retrieved by accessing a single table cache line.

Although prior work mentions this approach [19, 64], they do not evaluate its performance impact on lookups. Ross demonstrated its ability to reduce the cost of inserts but does not present data on its effect on lookups, instead opting to compare his latency-optimized lookup algorithm that always fetches f buckets to other open addressing methods and chaining [64]. Erlingsson et al. use the first-fit heuristic, but their results focus on the number of buckets accessed on insertions and feasible load factors for differing values of f and S (number of slots per bucket) and not the heuristic's impact on lookup performance [19]. For the sake of completeness, we evaluate both the load-balancing and first-fit heuristics in Section 8.

One consequence of using first-fit is that, because it less evenly balances the load across buckets, once the table approaches capacity, a few outliers repeatedly hash to buckets that are already full, necessitating long, cuckoo displacement chains when only 2 hash functions are used. Whereas we were able to implement the insertion routine for the load-balancing baseline and attain high load factors by relocating at most one or two elements, the first-fit heuristic prompted us to implement a serial version of the BFS approach described by Li et al. [47] because finding long displacement chains becomes necessary for filling the table to a comparable level. One solution to reducing these long chains is to use more hash functions. However, for BCHTs, this increases both the average- and worst-case lookup costs because each item can now appear in more distinct buckets. In the sections that follow, we demonstrate that these tradeoffs can be effectively circumvented with our technique and that there are additional benefits such as fast negative lookups.

4 Horton Tables

Horton tables are an extension of bucketized cuckoo hash tables that largely resolve the data movement issues of their predecessor when accessing buckets during lookups. They use two types of buckets (Figure 2): one is an unmodified BCHT bucket (**Type A**) and the other bucket flavor (**Type B**) contains additional in-bucket metadata to track elements that primarily hash to the bucket but have to be remapped due to insufficient capacity. All buckets begin as Type A and transition to Type B once they overflow, enabling the aforementioned tracking of displaced elements. This ability to track all remapped items at low cost, both in terms of time and space, permits the optimizations that give Horton tables their performance edge over the prior state of the art.

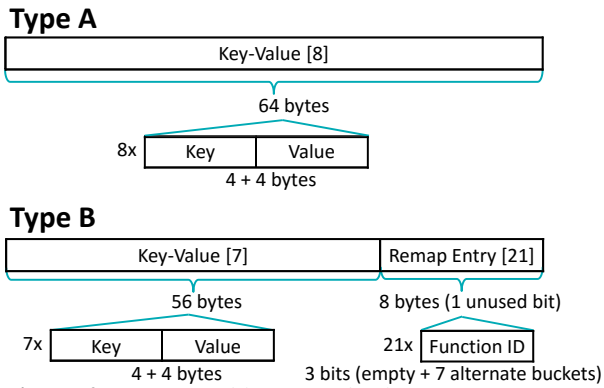


Figure 2: *Horton tables use 2 bucket variants: Type A (an unmodified BCHT bucket) and Type B (converts final slot into remap entries)*

Horton tables use $H_{primary}$, the **primary hash function**, to hash the vast majority of elements so that most lookups only require one bucket access. When inserting an item $KV = (K,V)$, it is only when the bucket at index $H_{primary}(K)$ cannot directly store KV that the item uses one of several **secondary hash functions** to remap the item. We term the bucket at index $H_{primary}(K)$ the **primary bucket** and buckets referenced by secondary hash functions **secondary buckets**. Additionally, **primary items** are key-value pairs that are directly housed in the bucket referenced by the primary hash function $H_{primary}$, and **secondary items** are those that have been remapped. There is no correlation between the bucket's type and its primacy; Type A and B buckets can simultaneously house both primary and secondary elements.

Type B buckets convert the final slot of Type A buckets into a **remap entry array**, a vector of k-bit[5] elements known as **remap entries** that encode the secondary hash function ID used to rehash items that cannot be accommodated in their primary bucket. Remap entries can take on one of 2^k different values, 0 for encoding an unused remap entry, and 1 to $2^k - 1$ for encoding which of the secondary hash functions R_1 to R_{2^k-1} was used to remap the items. To determine the index at which a remapped element's remap entry is stored, a tag hash function known as H_{tag} is computed on the element's key which maps to a spot in the remap entry array.

Remap entries are a crucial innovation of Horton tables, as they permit all secondary items to be tracked so that at most one primary and sometimes one secondary hash function need to be evaluated during table lookups regardless of whether (1) the lookup is positive or negative and (2) how many hash functions are used to rehash secondary items. At the same time, their storage is compactly allocated directly within the hash table bucket that overflows, boosting the locality of their accesses while still permitting high table load factors.

[5] k could range from 1 to the width of a key-value pair in bits, but we have found $k = 3$ to be a good design point.

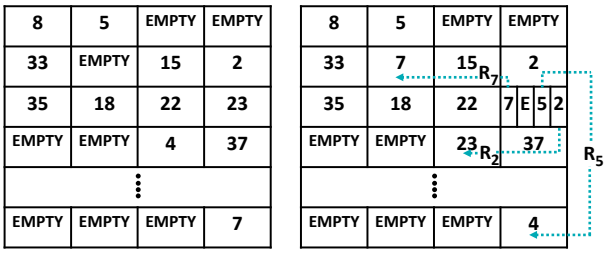


Figure 3: *Comparison of a bucketized cuckoo hash table (L) and a Horton table (R). E = empty remapy entry.*

With Horton tables, most lookups only require touching a single bucket and a small minority touch two. At the same time remap entries typically use at most several percent of the table's capacity, leaving sufficient free space for Horton tables to achieve comparable load factors to BCHTs.

Figure 2 shows the Type A and Type B bucket designs given 4-byte keys and values and 8-slot buckets. The bucket type is encoded in one bit of each bucket. For Type A buckets, this costs us a bit from just one of the value fields (now 31 bits). For Type B buckets, we encode 21 3-bit remap entries into a 64-bit slot, so we have a bit to spare already. If we have a value that requires all 32 bits in the last slot of a Type A bucket, we can move it to another slot in this bucket or remap to another bucket.

Because Type B buckets can house fewer elements than Type A buckets, Type A buckets are used as much as possible. It is only when a bucket has insufficient capacity to house all primary items that hash to it that it is converted into a Type B bucket, a process known as **promotion**. To guarantee that elements can be found quickly, whenever possible we enforce that primary elements are not displaced by secondary items. This policy ensures both that more buckets remain Type A buckets and that more items are primary elements.

4.1 A Comparison with BCHTs

Figure 3 shows a high-level comparison of Horton tables with an $f = 2$, traditional BCHT that stores the same data. Buckets correspond to rows and slots to individual cells within each row. In the Horton table (right), each item maps to its primary bucket by default. Bucket 2 (zero indexing) has been **promoted** from Type A to Type B because its 4 slots are insufficient to directly house the 6 key-value pairs that $H_{primary}$ has hashed there: 35, 18, 22, 7, 23, and 4. Because there is insufficient space to house 7, 23, and 4 directly in Bucket 2, they are remapped with hash functions R_7, R_2, and R_5, respectively, and the function IDs are stored directly in the remap entry array at the indices referenced by $H_{tag}(7) = 0$, $H_{tag}(23) = 3$, and $H_{tag}(3) = 2$. If we contrast the Horton table and the associated cuckoo hash table, we find that, of the shown buckets, the Horton table has a lookup cost of 1 for elements 8, 5, 33, 15, 2, 35, 18, 22, and 37 and a lookup cost of 2 for 7, 23, and 4, which averages out to 1.25. By contrast the bucketized cuckoo hash table has an expected lookup cost of 1.5 [20, 71] or 2.0 [47, 64], depending on the implementation.

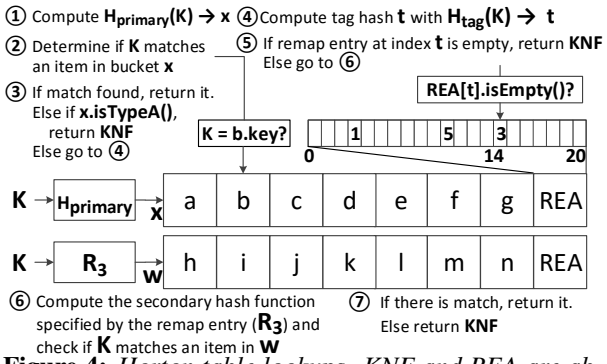


Figure 4: *Horton table lookups. KNF and REA are abbreviations for key not found and remap entry array.*

5 Algorithms

In this section, we describe the algorithms that we use to look up, insert, and delete elements in Horton tables. We begin each subsection by detailing the functional components of the algorithms and then, where relevant, briefly outline how each algorithm can be efficiently implemented using SIMD instructions.

5.1 Lookup Operation

Our lookup algorithm (Figure 4) works as follows. ① Given a key K, we first compute $H_{primary}(K)$, which gives us the index of K's primary bucket. ② We next examine the first $S - isTypeB()$ slots of the bucket, where $isTypeB()$ returns 1 if Bucket x is Type B and 0 if it is Type A by checking the bucket's most significant bit. ③ If the key is found, we return the value. In the case that the key is not found and the bucket is Type A, then the element cannot appear in any other bucket, and so we can declare the key not found.

④ If, however, the bucket is Type B, then we must examine the remap entry array when the item is not found in the first $S - 1$ slots of K's primary bucket (Bucket x). We first compute $H_{tag}(K)$ to determine the index into the remap entry array (shown as $t = 14$ in the figure). ⑤ If the value at that slot is 0, which signifies empty, then we declare the key not found because the key cannot appear in any other bucket. However, if the remap entry is set, then ⑥ we evaluate the secondary function R_i specified by the remap entry (R_3 in Figure 4) on a combination of the primary bucket and remap entry index (see Section 5.3.4) to get the index of the secondary bucket (Bucket w). We then compare K to the first $S - isTypeB()$ elements of w. ⑦ If we get a match, then we return it. If we do not get a match, then because an element is never found outside of its primary bucket or the secondary bucket specified by its remap entry, then we declare the key not found. It cannot be anywhere else.

5.2 SIMD Implementation of Lookups

Our approach leverages bulk processing of lookups and takes a large array of keys as input and writes out a corresponding array of retrieved values as output. We implement a modified version of Zhang et al.'s lookup algorithm [71], which virtualizes the SIMD unit into groups of S lanes (akin to simple cores that co-execute the same instruction stream) and assigns each

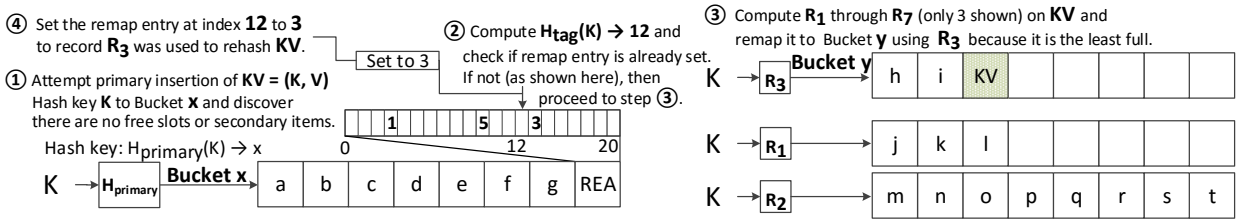


Figure 5: *Common execution path for secondary inserts. REA is an abbreviation of remap entry array.*

group a different bucket to process. When an element is found in the first $S - isTypeB()$ slots, we write the value out to an in-cache buffer. For the minority of lookups where more processing is necessary, e.g. computing the tag hash, indexing into the remap entry array, computing the secondary hash function, and searching the secondary bucket, we maintain a series of additional in-cache buffers where we enqueue work corresponding to these less frequent execution paths. When there is a SIMD unit's worth of work in a buffer, we dequeue that work and process it. Once a cache line worth of contiguous values have been retrieved from the hash table, we write those values back to memory in a single memory transaction.[6]

5.3 Insertion Operation

The primary goal of the insertion algorithm is to practically guarantee that lookups remain as fast as possible as the table's load factor increases. To accomplish this, we enforce at all levels the following guidelines:

1. Secondary items never displace primary items.
2. Primary items may displace both primary and secondary items.
3. When inserting into a full Type A bucket, only convert it to Type B if a secondary item in it cannot be remapped to another bucket to free up a slot.[7]

These guidelines ensure that as many buckets as possible remain Type A, which is important because converting a bucket from Type A to Type B can have a cascading effect: both the evicted element from the converted slot and the element that caused the conversion may map to other buckets and force them to convert to Type B as well. Further, Type B buckets house fewer elements, so they decrease the maximum load factor of the table and increase the expected lookup cost.

5.3.1 Primary Inserts

Given a key-value pair KV to insert into the table, if the primary bucket has a spare slot, then insertion can proceed by assigning that spare slot to KV. Spare slots can occur in Type A buckets as well as Type B buckets where a slot has been freed due to a deletion, assuming that Type B buckets do not atomically pull items back in from remap entries when slots become available. For the primary hash function, we use one of Jenkins' hash functions that operates on 32-bit inputs and produces 32-bit outputs [35]. The input to the function is the key, and we mod the output by the number of buckets to select a bucket to map the key to.

In the case where the bucket is full, we do not immediately attempt to insert KV into one of its secondary buckets but first search the bucket for secondary elements. If we find a secondary element KV' that can be remapped without displacing primary elements in other buckets, then we swap KV with KV' and rehash KV' to another bucket. Otherwise, we perform a secondary insert (see Sections 5.3.2, 5.3.3, and 5.3.4).

5.3.2 Secondary Inserts

We make a secondary insert when the item that we want to insert, $KV = (K,V)$, hashes to a bucket that is full and in which all items already stored in the bucket are primary. Most of the time, secondary inserts occur when an element's primary bucket has already been converted to Type B (see Section 5.3.3 and Figure 6 for the steps for converting from Type A to B); Figure 5 shows the most common execution path for a secondary insert.

① We first determine that a primary insert is not possible. ② We then compute the tag hash function on the key. If the remap entry at index $H_{tag}(K)$ is not set, we proceed to step ③. Otherwise, we follow the remap entry collision management scheme presented in Section 5.3.4 and Figure 7. ③ At this point, we need to find a free cell in which to place KV. We check each candidate bucket referenced by the secondary hash functions R_1 through R_7, and we place the remapped element in the bucket with least load, Bucket y in Figure 5. Alternatively, we could have placed KV into the candidate bucket with the first free slot – we chose the load-balancing approach because it reduced the prevalence of expensive cuckoo chains for relocating elements. ④ Lastly, we update the remap entry to indicate which secondary hash function was used. In this example, R_3 was used and H_{tag} on K evaluated to 12, so we write 3 in the 12th slot of the remap entry array of x, KV's primary bucket.

5.3.3 Conversion from Type A to Type B

Figure 6 shows the series of steps involved for converting a bucket from Type A to Type B. ① – ② If there are no secondary elements that can be displaced in the primary bucket, then the bucket evicts one of the items (h) to make room for the remap entry array, ③ – ⑤ rehashes it to another bucket, and ⑥ – ⑦ then proceeds by rehashing the element that triggered the conversion. As in the algorithm in Section 5.3.2, we attempt to relocate both items to their least loaded candidate buckets. ⑧ – ⑩ Once moved, the numerical identifier of each

[6] A simpler algorithm can be used when S is a multiple of the number of lanes, as all lanes within a SIMD unit process the same bucket.

[7] An item's primacy can be detected by evaluating $H_{primary}$ on its key. If the output matches the index where it is stored, then it is primary.

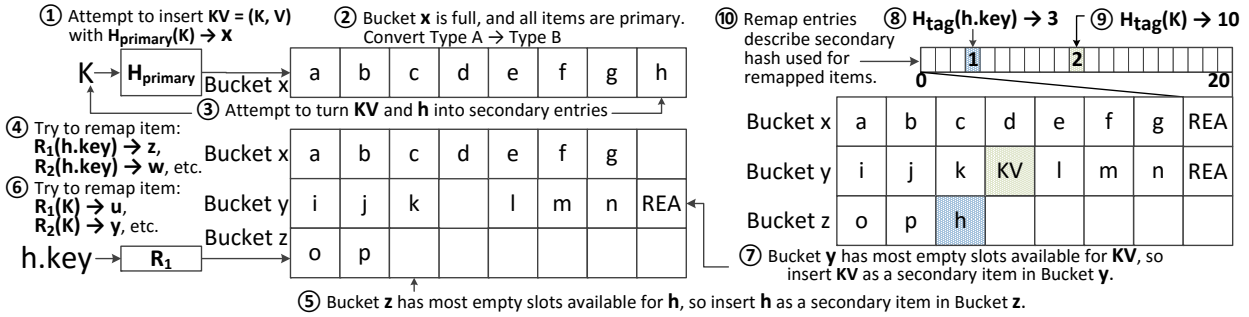


Figure 6: *Steps for converting from Type A to Type B. REA is an abbreviation of remap entry array.*

secondary hash function that remapped each of the two items (KV and h) is stored in the remap entry array of the primary bucket at the index described by H_{tag} of each key.

5.3.4 Remap Entry Collision Management

A major challenge of the remap entry approach is when two or more items that require remapping map to the same remap entry. Such collisions can be accommodated if all items that share a remap entry are made to use the same secondary hash function. However, if the shared secondary hash function takes the key as input, it will normally map each of the conflicting items to a different bucket. While this property poses no great challenge for lookups or insertions, it makes deletions of remap entries challenging because without recording that a remap entry is shared, we would not know whether another item is still referenced by it. Rather than associating a counter with each remap entry to count collisions as is done in counting Bloom filters [13, 22], we instead modify the secondary hash function so that items that share a remap entry map to the same secondary bucket. Since they share the same secondary bucket, we can check if it is safe to delete a remap entry on deletion of an element KV that the entry references by computing the primary hash function on each element in KV's secondary bucket. If none of the computed hashes reference KV's primary bucket for any of the elements that share KV's secondary bucket, then the remap entry can be deleted.

To guarantee that items that share remap entries hash to the same secondary bucket, we hash on a combination of the primary bucket index and the implicit tag as computed by $H_{tag}(key)$. Since this tuple uniquely identifies the location of each remap entry, we can create a one-to-one function from tuples to unique secondary hash function inputs, shown in Equation 1, where i is a number that uniquely identifies each secondary hash function and which ranges from 1 to $2^k - 1$ for k-bit remap entries (e.g. R_3 is the third secondary function out of 7 when k is 3), H_{L1} and H_{L2} are hash functions, k_{sec} is the secondary key derived from the tuple, and n is the number of remap entries per remap entry array. The uniqueness of these tuples as inputs is important for reducing collisions.

$$R_i(k_{sec}) = (H_{L1}(k_{sec}) + H_{L2}(k_{sec}, i)) \% \ Total\ Buckets$$
$$where\ k_{sec}(bucket\ index, tag) = bucket\ index * n + tag \quad (1)$$

By modifying the characteristics of H_{L1} and H_{L2}, we are able to emulate different hashing schemes. We employ modified double hashing by using Jenkins' hash [35] for H_{L1} and Equation 2 for H_{L2} where KT is a table of 8 prime numbers. We found this approach preferable because it makes it inexpensive to compute all of the secondary hash functions, reduces clustering compared to implementing H_{L2} as linear probing from H_{L1}, and, as Mitzenmacher proved, there is no penalty to bucket load distribution versus fully random hashing [55].

$$H_{L2}(k_{sec}, i) = KT[k_{sec} \% 8] * i \quad (2)$$

Sometimes the secondary bucket cannot accommodate additional elements that share the remap entry array. If so, we swap the item that we want to insert with another element from its primary bucket that can be rehashed. Because both elements in the swap are primary elements, this swap does not adversely affect the lookup rate.

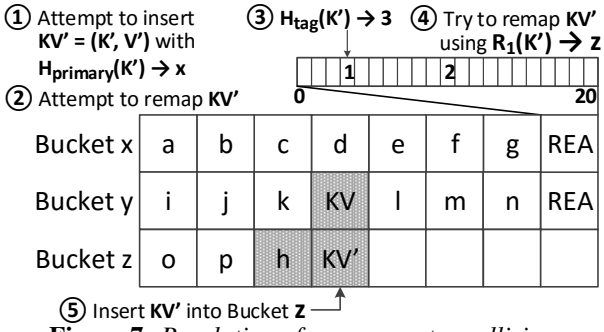


Figure 7: *Resolution of a remap entry collision*

Figure 7 presents a visual depiction of a remap entry collision during insertion that is resolved by having the new item map to the same bucket as the other items referenced by the remap entry. It continues from the example in Figure 6 and follows with inserting a new item KV'. ① When inserting KV', we first check for a free slot or an element that can be evicted from Bucket x because it is a secondary item when in x. ② If no such item exists, then we begin the process of remapping KV' to another bucket. ③ We first check the remap entry, and if it is set, ④ we proceed to the bucket it references, z in Figure 7. ⑤ We check for a free slot or a secondary item in z that can be displaced. If it is the former, we immediately insert KV'. Otherwise, we recursively evict elements until a free slot is found within a certain search tree height. Most of the time, this method works, but when it does not, we resort to swapping KV' with another element in its primary bucket that can be recursively remapped to a secondary bucket.

5.4 Deletion Operation

Deletions proceed by first calculating $H_{primary}$ on the key. If an item is found in the primary bucket with that key, that item is deleted. However, if it is not found in the primary bucket, and the bucket is Type B, then we check to see whether the remap entry is set. If it is, then we calculate the secondary bucket index and examine the secondary bucket. If an item with a matching key is found, then we delete that item. To determine whether we can delete the remap entry, we check to see if additional elements in the secondary bucket have the same primary bucket as the deleted element. If none do, we remove the remap entry.

5.4.1 Repatriation of Remapped Items

On deletions, slots that were previously occupied become free. In Type A buckets, there is no difference in terms of lookups regardless of how many slots are free. However, with Type B buckets, if a slot becomes free, that presents a performance opportunity to move a remapped item back into its primary bucket, reducing its lookup cost from 2 to 1 buckets. Similarly, if a Type B bucket has a combined total of fewer than $S+1$ items stored in its slots or remapped via its remap entries, it can be upgraded to a Type A bucket, which allows one more item to be stored and accessed with a single lookup in the hash table. Continual repatriation of items is necessary for workloads with many deletes to maximize lookup throughput and the achievable load factor. Determining when best to perform this repatriation, either via an eager or lazy heuristic, is future work.

6 Feasibility and Cost Analysis

In this section, we investigate the feasibility of using remap entries, the associated costs in terms of storage overhead, and the expected cost of both positive and negative hash table lookups.

6.1 Modeling Collisions

One of the most important considerations when constructing a Horton table is that each bucket should be able to track all items that initially hash to it using the primary hash function $H_{primary}$. In particular, given a hash table with B_T buckets and n inserted items, we want to be able to compute the expected number of buckets that have exactly x elements hash to them, for each value of x from 0 to n inclusive. By devising a model that captures this information, we can determine how many remap entries are necessary to facilitate the remapping and tracking of all secondary items that overflow their respective primary buckets.

If we assume that $H_{primary}$ is a permutation (i.e., it is invertible and the domain is the codomain), and that it maps elements to bins in a fashion that is largely indistinguishable from sampling a uniform random distribution, then given a selection of random keys and a given table size, we can precisely compute the expected number of buckets to which $H_{primary}$ maps exactly x key-value pairs by using a Poisson distribution based model [36]. The expected number of buckets with precisely x such elements, B_x, is given by Equation 3.

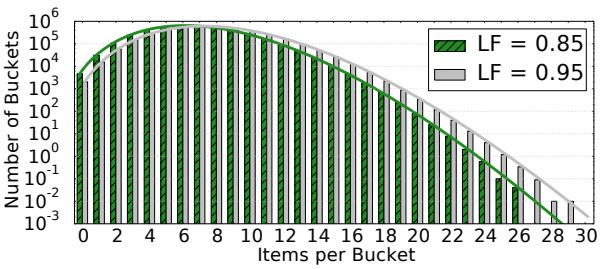


Figure 8: *Histogram of the number of buckets to which $H_{primary}$ assigns differing amounts of load in elements for two load factors. Curves represent instantiations of Equation 3 and bars correspond to simulation.*

$$
\begin{aligned}
B_x(\lambda,x) &= Total\ Buckets * P(\lambda,x) \\
&where\ P(\lambda,x) = \frac{e^{-\lambda}\lambda^x}{x!} \\
&where\ \lambda = Load\ Factor * Slots\ Per\ Bucket \\
&i.e.,\ \lambda = \frac{Elements\ Inserted}{Total\ Buckets}
\end{aligned}
\tag{3}
$$

The parameter λ is the mean of the distribution. Given a load factor, the average number of items that map to a bucket is the product of the load factor and the slots per bucket. Figure 8 coplots the results of a bucketized hash table simulation with results predicted by the analytical model given a hash table with 2^{22} 8-slot buckets. In our simulation, we created n unique keys in the range $[0,\ 2^{32}-1]$ using a 32-bit Mersenne Twister pseudorandom number generator [51] and maintained a histogram of counts of buckets with differing numbers of collisions. We found little to no variation in results with different commonly utilized hash functions (e.g., CityHash, SpookyHash, Lookup3, Wang's Hash, and FNV [26, 35, 59]). Therefore, we show only the results using one of Jenkins' functions that maps 32-bit keys to 32-bit values. Figure 8 shows a close correlation between the simulation results and Equation 3 for two load factors. Bars correspond to simulation results and curves to Equation 3. In each case, the model very closely tracks the simulation.

A high-level conclusion of the model is that with billions of keys and 8-slot buckets, there is a non-trivial probability that a very small subset of buckets will have on the order of 30 keys hash to them. This analysis informs our decision to use 21 remap entries per remap entry array and also the need to allow multiple key-value pairs to share each remap entry in order to reduce the number of remap entries that are necessary per bucket.

6.2 Modeling Remap Entry Storage Costs

In our hash table design, each promoted bucket trades one slot for a series of remap entries. To understand the total cost of remap entries, we need to calculate what percentage of buckets are Type A and Type B, respectively. For any hash table with S slots per bucket, Type A buckets have no additional storage cost, and so they do not factor into the storage overhead. Type B buckets on the other hand convert one of their S slots, i.e. $1/S$ of their usable storage, into a series of remap entries. Thus the expected space used by remap entries O_{re}, on a scale of

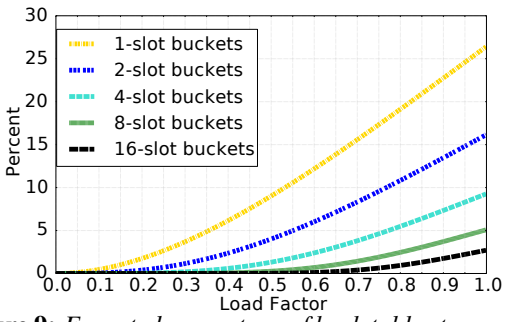


Figure 9: *Expected percentage of hash table storage that goes to remap entries as the load factor is varied*

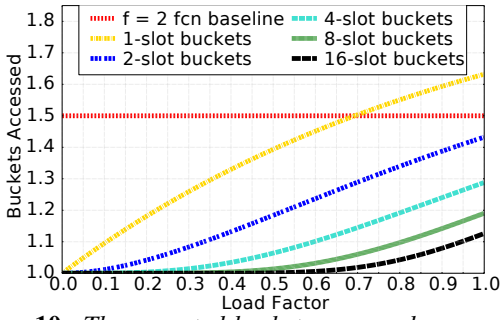


Figure 10: *The expected buckets accessed per positive lookup in a Horton table vs. a baseline BCHT that uses two hash functions*

0 (no remap entries) to 1 (entire table is remap entries), is the product of the fraction of Type B buckets and the consumed space $1/S$ (see Equation 4). For simplicity, we assume that each item that overflows a Type B bucket is remappable to a Type A bucket and that these remaps do not cause Type A buckets to become Type B buckets. This approximation is reasonable for two reasons. First, many hash functions can be used to remap items, and second, secondary items are evicted and hashed yet again, when feasible, if they prevent an item from being inserted with the primary hash function $H_{primary}$.

$$O_{re} = \frac{1}{S}\sum_{x=S+1}^{n} P(\lambda,x) \tag{4}$$

Figure 9 shows the expected percentage of hash table storage that goes to remap entries when varying the number of slots per bucket as well as the load factor. As the remap entries occupy space, the expected maximum load factor is strictly less than or equal to $1-O_{re}$. We see that neither 1 slot nor 2 slots per bucket is a viable option if we want to achieve load factors exceeding 90%. Solving for the expected bound on the load factor, we find that 4-, 8-, and 16-slot hash tables are likely to achieve load factors that exceed 91, 95, and 96%, respectively, provided that the remaining space not consumed by remap entries can be almost entirely filled.

6.3 Modeling Lookups

The expected average cost of a positive lookup is dependent on the percentage of items that are first-level lookups, the percentage of items that are second-level lookups, and the associated cost of accessing remapped and non-remapped items. For a bucket with S slots, if $x > S$ elements map to that bucket, $x - S + 1$ elements will need to be remapped, as one of those slots now contains remap entries. In the case where $x \leq S$, no elements need to be remapped from that bucket. The fraction of items that require remapping, I_{remap}, is given by Equation 5, and the fraction that do not, $I_{primary}$, is given by Equation 6. As stated previously, lookups that use $H_{primary}$ require searching one bucket, and lookups that make use of remap entries require searching two. Using this intuition, we combine Equations 5 and 6 to generate the expected positive lookup cost given by Equation 7. Since $I_{primary}$ is a probability, and $1 - I_{remap}$ is equivalent to $I_{primary}$, we can simplify the positive lookup cost to 1 plus the expected fraction of lookups that are secondary. Intuitively, Equation 7 makes sense: 100% of lookups need to access the primary bucket. It is only when the item has been remapped that a second bucket needs to be searched.

$$I_{remap} = \frac{\sum_{x=S+1}^{n}(x - S + 1) * P(\lambda,x)}{\lambda} \tag{5}$$

$$I_{primary} = \frac{\sum_{x=1}^{S}(x) * P(\lambda,x) + \sum_{x=S+1}^{n}(S-1) * P(\lambda,x)}{\lambda} \tag{6}$$

$$\begin{aligned} Positive\ Lookup\ Cost &= I_{primary} + 2I_{remap} \\ &= 1 + I_{remap} \end{aligned} \tag{7}$$

Figure 10 shows the expected positive lookup cost in buckets for 1-, 2-, 4-, 8-, and 16-slot bucket designs. Like before, buckets with more slots are better able to tolerate collisions. Therefore, as the number of slots per bucket increases for a fixed load factor, so does the ratio of Type A buckets to total buckets, which reduces the number of second-level lookups due to not needing to dereference a remap entry. In the 1- and 2-slot bucket cases, the benefit of remap entries is less pronounced but is still present. For the 1-slot case, there is a point at LF = 0.70 where we expect a baseline bucketized cuckoo hash table to touch fewer buckets. However, this scenario is not a fair comparison as, for a baseline BCHT with 1-slot buckets and two functions, the expected load factor does not reach 70%. To reach that threshold, many more hash functions would have to be used, increasing the number of buckets that must be searched. For the 4-, 8-, and 16-slot cases, we observe that the expected lookup cost is under 1.1 buckets for hash tables that are up to 60% full, and that even when approaching the maximum expected load factor, the expected lookup cost is less than 1.3 buckets. In the 8-slot and 16-slot cases, the expected costs at a load factor of 0.95 are 1.18 and 1.1 buckets, which represents a reduced cost of 21% and 27%, respectively, over the baseline.

The cost of a negative lookup follows similar reasoning. On a negative lookup, the secondary bucket is only searched on a false positive tag match in a remap entry. The expected proportion of negative lookups that exhibit tag aliasing, I_{alias}, is the product of the fraction of Type B buckets and the mean fraction of the tag space that is utilized per Type B bucket (Equation 8). In the implicit tag scheme, for a 64-bit remap entry array with 21 3-bit entries, the tag space is defined as the set $\{i \in \mathbb{N} | \ 0 \leq i \leq 20\}$ and has a tag space cardinality, call it C_{tag}, of 21. Alternatively, with explicit t-bit tags, C_{tag} would be 2^t minus any reserved values for designating

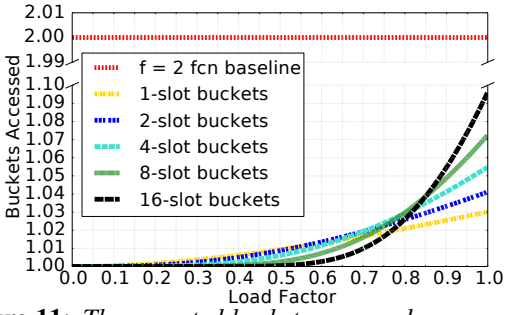


Figure 11: *The expected buckets accessed per negative lookup in a Horton table vs. a baseline BCHT that uses two hash functions*

states such as `empty`. For our model, we assume that there is a one-to-one mapping between remapped items and remap entries (i.e., each remap entry can only remap a single item). We further assume that conflicts where multiple items map to the same remap entry can be mitigated with high probability by swapping the element that would have mapped to an existing remap entry with an item stored in one of the slots that does not map to an existing remap entry, then rehashing the evicted element, and finally initializing the associated remap entry. These assumptions allow for at most $S-1+C_{tag}$ elements to be stored in or remapped from a Type B bucket.

$$I_{alias} = \frac{\sum_{x=S+1}^{S-1+C_{tag}}((x-S+1)*P(\lambda,x))}{C_{tag}} \quad (8)$$

$$\begin{aligned} Negative\ Lookup\ Cost &= I_{no\ alias} + 2I_{alias} \\ &= 1 + I_{alias} \end{aligned} \quad (9)$$

Like before, we can simplify Equation 9 by observing that all lookups need to search at least one bucket, and it is only on a tag alias that we search a second one. Because secondary buckets are in the minority and the number of remapped items per secondary bucket is often small relative to the tag space cardinality, the alias rate given in Equation 8 is often quite small, meaning that negative lookups have a cost close to 1.0 buckets.

In Figure 11, we plot the expected negative lookup cost for differing numbers of slots per bucket under progressively higher load factors with a tag space cardinality of 21. In contrast to positive lookups, adding more slots has a tradeoff. At low load factors, a table with more slots has a smaller proportion of elements that overflow and less Type B buckets, which reduces the alias rate. However, once the buckets become fuller, having more slots means that buckets have a greater propensity to have more items that need to be remapped, which increases the number of remap entries that are utilized. However, despite these trends, we observe that for 1-, 2-, 4-, 8-, and 16-slot buckets, aliases occur less than 8% of the time under feasible load factors, yielding an expected, worst-case, negative lookup cost of 1.08 buckets. Thus we expect Horton tables to reduce data movement on negative lookups by 46 to 50% versus an $f = 2$ BCHT.

7 Experimental Methodology

We run our experiments on a machine with a 4-core AMD A10-7850K with 32GB of DDR3 and an AMD Radeon™ R9-290X GPU with a peak memory bandwidth of 320 GB/s and 4GB of GDDR5. The L2 cache of the GPU is 1 MiB, and each of the 44 compute units has 16 KiB of L1 cache. Our system runs Ubuntu 14.04LTS with kernel version 3.16. Results for performance metrics are obtained by aggregating data from AMD's CodeXL, a publicly available tool that permits collecting high-level performance counters on GPUs when running OpenCL™ programs.

For the performance evaluation, we compare the lookup throughput of the load-balanced baseline, first-fit, and Horton tables. Our baseline implementation is most similar to Mega-KV [71], where the greatest difference is that we only use a single hash table rather than multiple independent partitions and use Jenkins' hash functions.

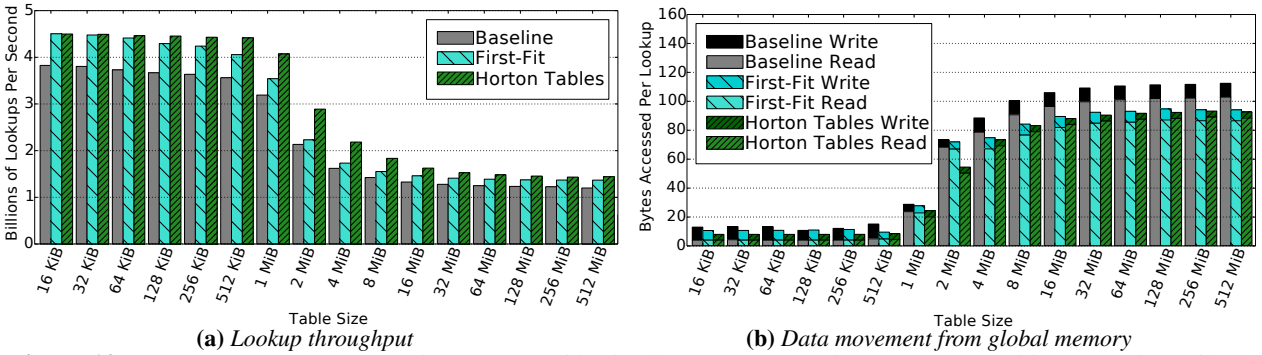
Insertions and deletions are implemented in C and run on the CPU. As our focus is on read-dominated workloads, we assume that insertion and deletion costs can largely be amortized and do not implement parallel versions. For each of the hash table variants, lookup routines are implemented in OpenCL [66] and run on the GPU, with each implementation independently autotuned for fairness of comparison. Toggled parameters include variable loop unrolling, the number of threads assigned to each compute unit, and the number of key-value pairs assigned to each group of threads to process. When presenting performance numbers, we do not count data transfer cost over PCIe because near-future integrated GPUs will have high-bandwidth, low-latency access to system memory without such overheads. This approach mirrors that of Polychroniou et al. [60] on the Xeon Phi [15].

As part of our evaluation, we validate the models presented in Section 6. We calculate the remap entry storage cost and lookup cost per item in terms of buckets by building the hash table and measuring its composition. Unless indicated elsewhere, we use 32-bit keys and values, 8-slot buckets and remap entry arrays with 21 3-bit entries. The probing array that we use for key-value lookups is 1 GiB in size. All evaluated hash tables are less than or equal to 512 MiB due to memory allocation limitations of the GPU; however, we confirmed that we were able to build heavily-loaded Horton tables with more than 2 billion elements without issue.

Keys and values for the table and the probing array are generated in the full 32-bit unsigned integer range using C++'s STL Mersenne Twister random integer generator that samples a pseudo-random uniform distribution [51]. We are careful to avoid inserting duplicate keys into the table, as that reduces the effective load factor by inducing entry overwrites rather than storing additional content. Since the probing array contains more keys than there are elements in the hash table, most keys appear multiple times in the probing array. We ensure that all such repeat keys appear far enough from one another such that they do not increase the temporal or spatial locality of accesses to the hash table. This approach is necessary to precisely lower bound the throughput of each algorithm for a given hash table size.

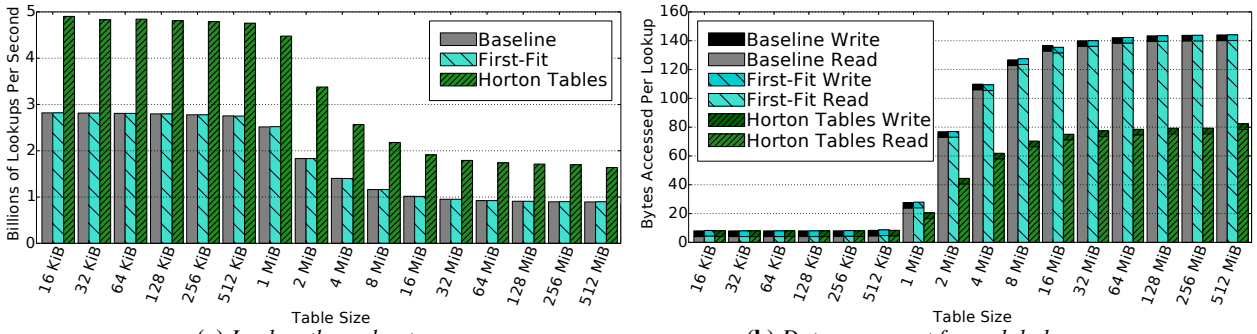
8 Results

In this section, we validate our models and present performance results. Figures 12a and 12b compare the lookup throughput and data movement (as seen by



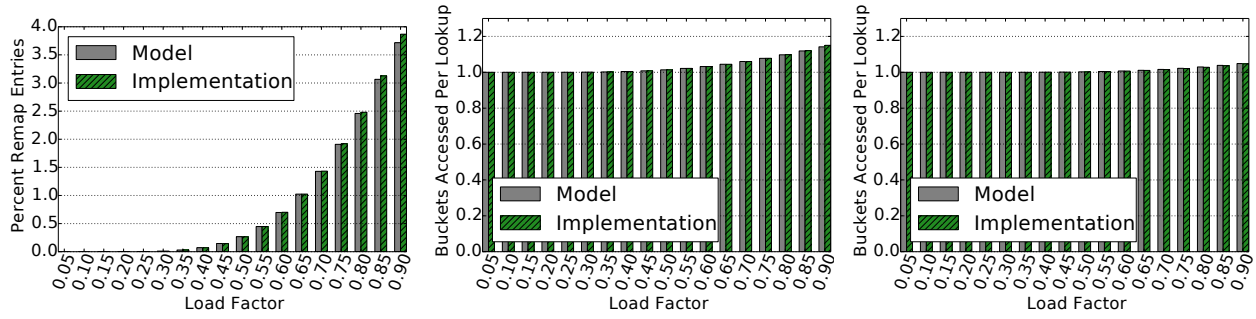
(a) *Lookup throughput* **(b)** *Data movement from global memory*

Figure 12: *Comparison of BCHTs with a Horton table (load factor = 0.9 and 100% of queried keys found in table)*



(a) *Lookup throughput* **(b)** *Data movement from global memory*

Figure 13: *Comparison of BCHTs with a Horton table (load factor = 0.9 and 0% of queried keys found in table)*



(a) *Remap entry storage cost* **(b)** *Positive lookup cost* **(c)** *Negative lookup cost*

Figure 14: *Validation of our models on an 8 MiB Horton table*

the global memory) between the load-balancing baseline (Section 3.3), BCHT with first-fit insert heuristic (Section 3.3) and Horton tables (Section 4) for tables from 16 KiB to 512 MiB in size. We see that Horton tables increase throughput over the baseline by 17 to 35% when all of the queried keys are in the table. In addition, they are faster than a first-fit approach, as Horton tables enforce primacy (Section 5.3, Guideline 1) on remapping of elements whereas first-fit does not. This discrepancy is most evident from 512 KiB to 8 MiB, where Horton tables are up to 29% faster than first-fit BCHTs.

These performance wins are directly proportional to the data movement saved. Initially, there is no sizeable difference in the measured data movement cost between the baseline, first-fit, and Horton tables, as the hash tables entirely fit in cache. Instead, the bottleneck to performance is the cache bandwidth. However, at around 1 MiB, the size at which the table's capacity is equal to the size of the last level cache (L2), the table is no longer fully cacheable in L2, and so it is at this point that the disparity in data movement between the three approaches becomes visible at the off-chip memory.

Figures 13a and 13b show the opposite extreme where none of the queried keys are in the table. In this case, Horton tables increase throughput by 73 to 89% over the baseline and first-fit methods because, unlike a BCHT, Horton tables can satisfy most negative searches with one bucket access. These results and those for positive lookups from Figures 12a and 12b align very closely with the reduction in data movement that we measured with performance counters. For a workload consisting entirely of positive lookups, baseline BCHTs access 30% more cache lines than Horton tables. At the opposite extreme, for a workload of entirely negative lookups, both first-fit and baseline BCHTs access 90% more hash table cache lines than Horton tables.

If we examine the total data movement, we find that both our BCHT and Horton table implementations move

an amount of data close to what our models project. At a load factor of 0.9, our model predicts 1.15 and 1.05 buckets accessed per positive and negative query, respectively. Since cache lines are 64 bytes, this cost corresponds to 74 and 67 bytes per query worth of data movement. On top of that, for each lookup query we have an additional 8 bytes of data movement for loading the 4-byte query key and 4 bytes for storing the retrieved value, which puts our total positive and negative lookup costs at 82 and 75 bytes, respectively. These numbers are within 10% of the total data movement that we observe in Figures 12b and 13b once the hash table is much larger than the size of the last-level cache. Similarly, we found that our models' data movement estimates for BCHTs were within similar margins of our empirical results.

Figures 14a, 14b, and 14c show that each of our models accurately capture the characteristics of our implementation. On average, our table requires fewer than 1.15 bucket lookups for positive lookups and fewer than 1.05 for negative lookups at a load factor of 0.9, and both have a cost of essentially 1.0 up to a load factor of 0.55. These results are a dramatic improvement over the current state of practice and validate the soundness of our algorithms to achieve high load factors without measurably compromising on the percentage of primary lookups.

9 Related Work

There has been a long evolution in hash tables. Two pieces of work that share commonality with our own are the cuckoo filter [21] and MemC3 [20]. MemC3 is a fast, concurrent alternative to Memcached [24] that uses a bucketized hash table to index data, with entries that consist of (1) tags and (2) pointers to objects that house the full key, value, and additional metadata. For them, the tag serves two primary functions: (1) to avoid polluting the cache with long keys on most negative lookups and (2) to allow variable-length keys. Tags are never used to avoid bringing additional buckets into cache. If the element is not found in the first bucket, the second bucket is always searched. Similarly, the cuckoo filter is also an $f = 2$ function, 4-slot bucketized cuckoo hash set that is designed to be an alternative to Bloom filters [10] that is cache friendly and supports deletions.

Another related work is Stadium Hashing [41]. Their focus is to have a fast hash table where the keys are stored on the GPU and the values in the CPU's memory. Unlike us they use a non-bucketized hash table with double hashing and prime capacity. They employ an auxiliary data structure known as a ticket board to filter requests between the CPU and GPU and also to permit concurrent put and get requests to the same table. Barber et al. use a similar bitmap structure to implement two compressed hash table variants [7].

The BCHT [19] combines cuckoo hashing and bucketization [25, 57, 58]. Another improvement to cuckoo hashing is by way of a stash—a small, software victim cache for evicted items [42].

Other forms of open addressing are also prevalent. Quadratic hashing, double hashing, Robin Hood hashing [14], and linear probing are other commonly used open addressing techniques [16]. Hopscotch hashing attempts to move keys to a preferred neighborhood of the table by displacing others [31]. It maintains a per-slot *hop information* field that is often several bits in length that tracks element displacements to nearby cells. By contrast, Horton tables only create remap entries when buckets exceed their baseline capacity.

Other work raises the throughput of concurrent hash tables by using lock free approaches [53, 67, 68] or fine-grain spinlocks [47]. Additional approaches attempt to fuse or use other data structures in tandem with hash tables to enable faster lookups [48, 61, 65].

In application, hash tables find themselves used in a wide of variety of data warehousing and processing applications. A number of in-memory key-value stores employ hash tables [20, 24, 32, 56, 71], and others accelerate key lookups by locating the hash table on the GPU [32, 33, 71]. Early GPU hash tables have been primarily developed for accelerating applications in databases, graphics and computer vision [2, 3, 27, 43, 45]. In in-memory databases, there has been significant effort spent on optimizing hash tables due to their use in hash join algorithms on CPUs [5, 6, 9, 12, 60], coupled CPU-GPU systems [28, 29, 38], and the Xeon Phi [37, 60].

This work on high-performance hash tables is complementary to additional research efforts that attempt to retool other indexes such as trees to take better advantage of system resources and new and emerging hardware [46, 49, 50, 69].

10 Conclusion

This paper presents the Horton table, an enhanced bucketized cuckoo hash table that achieves higher throughput by reducing the number of hardware cache lines that are accessed per lookup. It uses a single function to hash most elements and can therefore retrieve most items by accessing a single bucket, and thus a single cache line. Similarly, most negative lookups can also be satisfied by accessing one cache line. These low access costs are enabled by remap entries: sparingly allocated, in-bucket records that enable both cache and off-chip memory bandwidth to be used much more efficiently. Accordingly, Horton tables increase throughput for positive and negative lookups by as much as 35% and 89%, respectively. Best of all, these improvements do not sacrifice the other attractive traits of baseline BCHTs: worst-case lookup costs of 2 buckets and load factors that exceed 95%.

11 Acknowledgements

The authors thank the reviewers for their insightful comments and Geoff Kuenning, our shepherd, for his meticulous attention to detail and constructive feedback. We were very impressed with the level of rigor that was applied throughout the review and revision process. We also thank our peers at AMD Research for their comments during internal presentations of the work and Geoff Voelker for providing us invaluable feedback that elevated the paper's quality.

References

[1] AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND WOOD, D. A. DBMSs on a Modern Processor: Where Does Time Go? In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)* (1999).

[2] ALCANTARA, D. A., SHARF, A., ABBASINEJAD, F., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., AND AMENTA, N. Real-Time Parallel Hashing on the GPU. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)* (2009).

[3] ALCANTARA, D. A., VOLKOV, V., SENGUPTA, S., MITZENMACHER, M., OWENS, J. D., AND AMENTA, N. Building an Efficient Hash Table on the GPU. In *GPU Computing Gems: Jade Edition*, W. W. Hwu, Ed. Morgan Kaufmann, 2011, ch. 4, pp. 39–54.

[4] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-Scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[5] BALKESEN, C., ALONSO, G., TEUBNER, J., AND ÖZSU, M. T. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. of the VLDB Endowment 7*, 1 (2013), 85–96.

[6] BALKESEN, C., TEUBNER, J., ALONSO, G., AND OZSU, M. T. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2013).

[7] BARBER, R., LOHMAN, G., PANDIS, I., RAMAN, V., SIDLE, R., ATTALURI, G., CHAINANI, N., LIGHTSTONE, S., AND SHARPE, D. Memory-Efficient Hash Joins. *Proc. of the VLDB Endowment 8*, 4 (2014), 353–364.

[8] BENDER, M. A., FARACH-COLTON, M., JOHNSON, R., KRANER, R., KUSZMAUL, B. C., MEDJEDOVIC, D., MONTES, P., SHETTY, P., SPILLANE, R. P., AND ZADOK, E. Don't Thrash: How to Cache Your Hash on Flash. *Proc. of the VLDB Endowment 5*, 11 (2012), 1627–1637.

[9] BLANAS, S., LI, Y., AND PATEL, J. M. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (2011).

[10] BLOOM, B. H. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[11] BONCZ, P. A., KERSTEN, M. L., AND MANEGOLD, S. Breaking the Memory Wall in MonetDB. *Communications of the ACM 51*, 12 (2008), 77–85.

[12] BONCZ, P. A., MANEGOLD, S., AND KERSTEN, M. L. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB)* (1999).

[13] BONOMI, F., MITZENMACHER, M., PANIGRAHY, R., SINGH, S., AND VARGHESE, G. An Improved Construction for Counting Bloom Filters. In *Proc. of the Annual European Symposium (ESA)* (2006).

[14] CELIS, P., LARSON, P.-Å., AND MUNRO, I. J. Robin Hood Hashing. In *Proc. of the IEEE Annual Symp. on Foundations of Computer Science (FOCS)* (1985).

[15] CHRYSOS, G., AND ENGINEER, S. P. Intel Xeon Phi Coprocessor (Codename Knights Corner). Presented at Hot Chips, 2012.

[16] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[17] DEWITT, D. J., KATZ, R. H., OLKEN, F., SHAPIRO, L. D., STONEBRAKER, M. R., AND WOOD, D. A. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (1984).

[18] DR. SEUSS. *Horton Hatches the Egg*. Random House, 1940.

[19] ERLINGSSON, U., MANASSE, M., AND MCSHERRY, F. A Cool and Practical Alternative to Traditional Hash Tables. In *Proc. of the Workshop on Distributed Data and Structures (WDAS)* (2006).

[20] FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2013).

[21] FAN, B., ANDERSEN, D. G., KAMINSKY, M., AND MITZENMACHER, M. D. Cuckoo Filter: Practically Better Than Bloom. In *Proc. of the ACM Int'l Conf. on Emerging Networking Experiments and Technologies (CoNEXT)* (2014).

[22] FAN, L., CAO, P., ALMEIDA, J., AND BRODER, A. Z. Summary Cache: a Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking (TON) 8*, 3 (2000), 281–293.

[23] FATAHALIAN, K., AND HOUSTON, M. A Closer Look at GPUs. *Communications of the ACM 51*, 10 (2008), 50–57.

[24] FITZPATRICK, B. Distributed Caching with Memcached. *Linux Journal 2004*, 124 (Aug. 2004), 5.

[25] FOTAKIS, D., PAGH, R., SANDERS, P., AND SPIRAKIS, P. Space Efficient Hash Tables with Worst Case Constant Access Time. In *Proc. of the Annual Symp. on Theoretical Aspects of Computer Science (STACS)* (2003).

[26] FOWLER, G., AND CURT NOLL, L. The FNV Non-Cryptographic Hash Algorithm. `http://tools.ietf.org/html/draft-eastlake-fnv-03`. Accessed: 2015-12-01.

[27] GARCÍA, I., LEFEBVRE, S., HORNUS, S., AND LASRAM, A. Coherent Parallel Hashing. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)* (2011).

[28] HE, B., YANG, K., FANG, R., LU, M., GOVINDARAJU, N., LUO, Q., AND SANDER, P. Relational Joins on Graphics Processors. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (2008).

[29] HE, J., LU, M., AND HE, B. Revisiting Co-Processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. of the VLDB Endowment 6*, 10 (2013), 889–900.

[30] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2011.

[31] HERLIHY, M., SHAVIT, N., AND TZAFRIR, M. Hopscotch Hashing. In *Proc. of the Int'l Symp. on Distributed Computing (DISC)* (2008).

[32] HETHERINGTON, T. H., O'CONNOR, M., AND AAMODT, T. M. MemcachedGPU: Scaling-up Scale-out Key-value Stores. In *Proc. of the ACM Symp. on Cloud Computing (SoCC)* (2015).

[33] HETHERINGTON, T. H., ROGERS, T. G., HSU, L., O'CONNOR, M., AND AAMODT, T. M. Characterizing and Evaluating a Key-value Store Application on Heterogeneous CPU-GPU Systems. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)* (2012).

[34] HOFSTEE, H. P. Power Efficient Processor Architecture and the Cell Processor. In *Proc. of the Int'l Symp. on High-Performance Computer Architecture (HPCA)* (2005).

[35] JENKINS, B. 4-byte Integer Hashing. `http://burtleburtle.net/bob/hash/integer.html`. Accessed: 2015-12-01.

[36] JENKINS, B. Some Random Theorems. `http://burtleburtle.net/bob/hash/birthday.html`. Accessed: 2015-12-01.

[37] JHA, S., HE, B., LU, M., CHENG, X., AND HUYNH, H. P. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. of the VLDB Endowment 8*, 6 (2015), 642–653.

[38] KALDEWEY, T., LOHMAN, G., MUELLER, R., AND VOLK, P. GPU Join Processing Revisited. In *Proc. of the Int'l Workshop on Data Management on New Hardware (DaMoN)* (2012).

[39] KECKLER, S. W., DALLY, W. J., KHAILANY, B., GARLAND, M., AND GLASCO, D. GPUs and the Future of Parallel Computing. *IEEE Micro*, 5 (2011), 7–17.

[40] KEMPER, A., AND NEUMANN, T. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2011).

[41] KHORASANI, F., BELVIRANLI, M. E., GUPTA, R., AND BHUYAN, L. N. Stadium Hashing: Scalable and Flexible Hashing on GPUs. In *Proc. of the Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)* (2015).

[42] KIRSCH, A., MITZENMACHER, M., AND WIEDER, U. More robust Hashing: Cuckoo Hashing with a Stash. *SIAM Journal on Computing 39*, 4 (2009), 1543–1561.

[43] KORMAN, S., AND AVIDAN, S. Coherency Sensitive Hashing. In *Proc. of the IEEE Int'l Conf. on Computer Vision (ICCV)* (2011).

[44] LEE, V. W., KIM, C., CHHUGANI, J., DEISHER, M., KIM, D., NGUYEN, A. D., SATISH, N., SMELYANSKIY, M., CHENNUPATY, S., HAMMARLUND, P., SINGHAL, R., AND DUBEY, P. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proc. of the Int'l Symp. on Computer Architecture (ISCA)* (2010).

[45] LEFEBVRE, S., AND HOPPE, H. Perfect Spatial Hashing. In *Proc. of the ACM SIGGRAPH Conf. and Exhibition on Computer Graphics and Interactive Techniques (SIGGRAPH)* (2006).

[46] LEVANDOSKI, J. J., LOMET, D. B., AND SENGUPTA, S. The Bw-Tree: A B-tree for New Hardware Platforms. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2013).

[47] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proc. of the European Conf. on Computer Systems (EuroSys)* (2014).

[48] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proc. of the ACM Symp. on Operating Systems Principles (SOSP)* (2011).

[49] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2014).

[50] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proc. of the European Conf. on Computer Systems (EuroSys)* (2012).

[51] MATSUMOTO, M., AND NISHIMURA, T. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[52] MCKEE, S. A. Reflections on the Memory Wall. In *Proc. of the ACM Int'l Conf. on Computing Frontiers (CF)* (2004).

[53] METREVELI, Z., ZELDOVICH, N., AND KAASHOEK, M. F. CPHash: A Cache-Partitioned Hash Table. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)* (2012).

[54] MITZENMACHER, M. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. on Parallel and Distributed Systems (TPDS) 12*, 10 (2001), 1094–1104.

[55] MITZENMACHER, M. Balanced Allocations and Double Hashing. In *Proc. of the ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)* (2014).

[56] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcache at Facebook. In *Proc. of the USENIX Symp. on Networked Systems Design and Implementation (NSDI)* (2013).

[57] PAGH, R., AND RODLER, F. F. Cuckoo Hashing. *Journal of Algorithms 51*, 2 (2004), 122–144.

[58] PANIGRAHY, R. Efficient Hashing with Lookups in Two Memory Accesses. In *Proc. of the ACM-SIAM Symp. on Discrete Algorithms (SODA)* (2005).

[59] PIKE, G., AND ALAKUIJALA, J. Introducing CityHash. `http://google-opensource.blogspot.com/2011/04/introducing-cityhash.html`. Accessed: 2015-12-01.

[60] POLYCHRONIOU, O., RAGHAVAN, A., AND ROSS, K. A. Rethinking SIMD Vectorization for In-Memory Databases. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD)* (2015).

[61] RAMABHADRAN, S., RATNASAMY, S., HELLERSTEIN, J. M., AND SHENKER, S. Prefix Hash Tree: An Indexing Data Structure Over Distributed Hash Tables. In *Proc. of the ACM Symp. on Principles of Distributed Computing (PODC)* (2004).

[62] RAMAN, V., ATTALURI, G., BARBER, R., CHAINANI, N., KALMUK, D., KULANDAISAMY, V., LEENSTRA, J., LIGHTSTONE, S., LIU, S., LOHMAN, G. M., MALKEMUS, T., MUELLER, R., PANDIS, I., SCHIEFER, B., SHARPE, D., SIDLE, R., STORM, A., AND ZHANG, L. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *Proc. of the VLDB Endowment 6*, 11 (2013), 1080–1091.

[63] RICHA, A. W., MITZENMACHER, M., AND SITARAMAN, R. The Power of Two Random Choices: A Survey of Techniques and Results. In *Handbook of Randomized Computing*, S. Rajasekaran, P. M. Pardalos, J. Reif, and J. Rolim, Eds., vol. 1. Kluwer Academic Publishers, 2001, ch. 9, pp. 255–304.

[64] ROSS, K. Efficient Hash Probes on Modern Processors. In *Proc. of the IEEE Int'l Conf. on Data Engineering (ICDE)* (2007).

[65] SONG, H., DHARMAPURIKAR, S., TURNER, J., AND LOCKWOOD, J. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proc. of the Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2005).

[66] STONE, J. E., GOHARA, D., AND SHI, G. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science & Engineering 12*, 3 (2010), 66–73.

[67] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Scalable Concurrent Hash Tables via Relativistic Programming. *ACM SIGOPS Operating Systems Review 44*, 3 (2010), 102–109.

[68] TRIPLETT, J., MCKENNEY, P. E., AND WALPOLE, J. Resizable, Scalable, Concurrent Hash Tables via Relativistic Programming. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)* (2011), p. 11.

[69] WU, X., XU, Y., SHAO, Z., AND JIANG, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *Proc. of the USENIX Annual Technical Conf. (USENIX ATC)* (2015).

[70] WULF, W. A., AND MCKEE, S. A. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News 23*, 1 (1995), 20–24.

[71] ZHANG, K., WANG, K., YUAN, Y., GUO, L., LEE, R., AND ZHANG, X. Mega-KV: A case for GPUs to Maximize the Throughput of In-Memory Key-Value Stores. *Proc. of the VLDB Endowment 8*, 11 (2015), 1226–1237.

Ginseng : Market-Driven LLC Allocation

Liran Funaro Orna Agmon Ben-Yehuda Assaf Schuster
Technion—Israel Institute of Technology
{funaro,ladypine,assaf}@cs.technion.ac.il

Abstract

Cloud providers must dynamically allocate their physical resources to the right client to maximize the benefit that they can get out of given hardware. Cache Allocation Technology (CAT) makes it possible for the provider to allocate last level cache to virtual machines to prevent cache pollution. The provider can also allocate the cache to optimize client benefit. But how should it optimize client benefit, when it does not even know what the client plans to do?

We present an auction-based mechanism that dynamically allocates cache while optimizing client benefit and improving hardware utilization. We evaluate our mechanism on benchmarks from the Phoronix Test Suite. Experimental results show that *Ginseng* for cache allocation improved clients' aggregated benefit by up to 42.8× compared with state-of-the-art static and dynamic algorithms.

1 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing providers rent computing resources wrapped as an infrastructure, i.e., a guest virtual machine (VM), to their clients. To compete in the tough market of cloud computing, providers must improve their clients' quality-of-service (QoS) while maintaining competitive pricing and reducing per-client management cost. Thus, better hardware utilization is necessary. New Intel technology that supports last level cache (LLC) allocation allows better cache utilization via cache partitioning.

Providers can utilize this new technology to guarantee clients' performance requirements by preventing applications from polluting each other's cache [32], as Intel intended [25]. Moreover, they can accommodate more clients' performance requirements by granting more LLC to those who benefit from it and preventing access to those who do not. This increases the provider's ability to consolidate the physical host.

Without any client performance information, the provider can only optimize guest performance according to host-known metrics, such as instructions-per-second (IPC), LLC hit-rate, LLC reads-count, and so forth [19, 41, 64]. The host does not know what the client's real benefit from more cache is, nor can it compare benefits that different clients draw from cache. For example, higher IPC does not necessarily indicate better performance, as the guest VM might just be polling on a spin-lock more quickly.

Moreover, a client may be willing to settle for poorer performance in exchange for a lower payment [5]. This might be the case, for example, when the guest VM of a performance-demanding client is running maintenance work in between important workloads every few seconds. Nevertheless, lacking the guests' current workload information, the host will try to improve its performance despite the lack of benefit to the client. This, in turn, may hinder the performance of other guests. It is therefore in the interest of both provider and client that clients pay only for the fine grained LLC they need, when they need it [1, 3, 7, 15, 45, 49]. Client satisfaction will thus be improved, as clients can pay less for the same performance but only when it is really needed, while providers will be able to improve hardware utilization.

However, real-world public cloud clients are *selfish*, *rational* economic entities. They will not let the provider know precisely how much benefit each quantity of cache ways would bring to it. They are *black-boxes*, and as such, unlike *white-box* clients [15, 22, 23, 47], they will not share their *true* private information with their provider unless it is in their own best interest to do so. For example, if the host allocates cache ways to guests who will derive the greatest benefit from it, each guest will claim that it has the most to gain from additional cache ways. Likewise, if the host allocates cache ways to guests who perform poorly, each guest will claim poor performance.

Even passive *black-box* measurements taken by the provider can be manipulated [19, 41, 64]. For example, a guest can fake cache misses by adding random instructions in the code that access random—non-cached—memory addresses. Such instructions will not delay the out-of-order-execution (OOOE) CPU as they are independent of the other instructions, and they will induce many cache misses.

In this paper we address the problem of how cloud providers should allocate cache among selfish black-box clients in light of the new cache allocation technology.

Our contribution towards a solution to this problem is *Ginseng* [4] for *cache allocation*, a market-driven auction system that maximizes the aggregated benefit of the guests in terms of the economic value they attribute to the desired allocation, using game-theoretic principles. This approach encourages even a *selfish* guest to bid for cache according to its *true* benefit. *Ginseng* was first introduced for memory allocation [4], and a similar, auction-based approach was used before for bandwidth allocation [38]. Furthermore, Amazon has been auctioning virtual machines since 2009 [2].

We evaluate *Ginseng* on benchmarks from the Phoronix Test Suite [37], which we classify according to their benefit from the cache. We show that *Ginseng* improves the aggregated economic benefit of guests from cache by up to 42.8× compared to the prevalent method used by today's cloud providers.

Our second contribution is an evaluation of the attributes which differentiate dynamic cache allocation from other resources: **(1)** As opposed to memory, cache does not have to be exclusively allocated and can be shared effectively. However, mutual trust is required to allow benefit for the sharing participants. **(2)** Unlike bandwidth—but much like memory—cache has to be warmed up before the guest can benefit from it. However—unlike memory pages—cache must be allocated consecutively, which induces more cache way transfers when the allocation is changed. Furthermore, Intel's allocation mechanism might fail to enforce frequent transfers of cache ownership. Thus, dynamic cache allocation might incur performance overhead. We address the question of when it is beneficial to share cache and when exclusive allocation is preferable, and we measure and analyze the overhead of frequently changing the allocation.

2 System Architecture

Ginseng is a market-driven cloud mechanism that allocates resources to guest virtual-machines by means of an auction. It is implemented for cloud hosts running the KVM hypervisor [35] but can work seamlessly on any other hypervisor. *Ginseng* for *cache allocation* controls the cache ways allocated to each guest using the *cache-driver* described in §3.

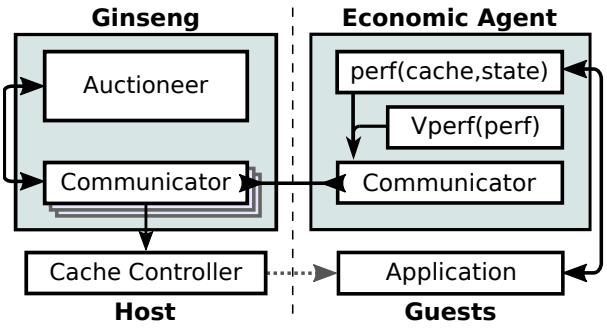


Figure 1: Ginseng system architecture

Ginseng has a host component and a guest component, as depicted in Figure 1. The host's Auctioneer uses the Vickrey-Clarke-Groves (VCG) auction [10, 17, 60], to be described in §4. The host's communicators communicate with the guests using the auction protocol, detailed in §5. It also instructs the cache controller how to allocate cache ways among the guests. The guest's economic agent bids on behalf of the client by stating a valuation for each number of cache ways. The guest component we implemented bids with a true valuation, as that is the best strategy for the guest [10, 17, 60], but *Ginseng* does not enforce any restrictions on the implementation and strategy of the guest's economic agent.

3 Cache Architecture

Intel's cache, and LLC in particular, stores data in granularity of cache lines that typically vary from 64 to 256 bytes, depending on the machine. The cache is organized in ways, each of which is a hash table, where the key is a hash value of the line's memory address and the value is the content of the cache line. Way locations that are designated to be filled by lines with the same keys are called a set.

When reading from a memory address or writing to it, the CPU first computes the address's set by using the hash function. Then the line is stored in this set on one of the cache ways. If the entire set is full, the least recently used (LRU) line in the set will be evicted and replaced.

When an application uses the cache exclusively, it will evict its own least recently used data. However, when several applications use the same cache, one might evict the other's cache lines and influence its performance. To prevent this, Intel's new cache allocation technology (CAT) allows cache partitioning. The API defines the notion of classes-of-service (COS), which determine a set of cache ways. When a hardware thread is assigned to a COS, it is only allowed to store new cache lines in the

ways determined by the COS. However, the COS does not limit reading from any of the ways in the cache.

Intel's API requires that the selected ways in each COS be consecutive. The API does not impose exclusivity, so a cache way can be used by more than one COS.

We experimented with new hardware (Haswell) that supports the new cache allocation technology. It supports only four COSes and requires a minimum allocation of two cache ways for each COS. However, more advanced architectures such as Broadwell will support a minimum of one cache-way allocation and 16 COSes [26].

Intel has already added support for CAT to the Linux kernel (tip) via the cgroups interface. However, at the time we experimented with the hardware, the modification was only available as a patch and was not stable enough. Therefore, we implemented our own user-level driver that allows control over the COSes and their assignment to CPUs. We implemented it in Python by writing directly to the model-specific registers (MSR) using rdmsr/wrmsr utilities for Linux. The driver code is available at `https://bitbucket.org/fonaro/cat-driver`.

3.1 Restricting LLC Access: The Pit

Preventing LLC access to some guests will allow us to allocate more cache ways to others who might benefit more from it. However, the cache allocation API does not allow LLC access to be restricted to specific guests. Instead, we assign them to a single COS we denote the *pit*. We allocate the *pit* the minimum number of cache ways allowed by the hardware (two for our hardware).

4 Cache Auction

Ginseng allocates cache efficiently because it uses a game-theoretic mechanism to elicit the guests' true benefit from cache. The host conducts rounds of cache auctions to adapt the allocation according to the guests' changing needs.

In *Ginseng*, each guest has a different, changing, private (secret) valuation of cache, which is expressed in dollars per second for each cache way allocation. Each way will be allocated exclusively. The guest derives its valuation by combining two private functions: performance as a function of cache ways (in performance units per second) and valuation of performance (in dollars per performance unit). By taking into account resource allocation and monetary worth, *Ginseng* is able to compare valuations, while the actual performance requirements are defined and controlled by the client [18].

As in *Ginseng* for memory allocation [4], we define the aggregate benefit from a cache allocation to all guests—their satisfaction from the auction results—using the game-theoretic measure of *social welfare*. The social welfare of an allocation is defined as the sum of all the guests' valuations of the cache they receive in the allocation.

Ginseng for *cache allocation* uses the VCG auction, which maximizes social welfare by encouraging even selfish participants with conflicting economic interests to inform the auctioneer of their true valuation of the auctioned goods. In VCG auctions, this is done by charging each participant for the damage it inflicts on other participants' social welfare, rather than directly for the goods it wins. VCG auctions are used, for example, in Facebook's repeated auctions [43], as well as in other settings.

The guest's valuation for each allocation of cache can be affected by its expected performance given its current state. However, it can also be affected by variables unrelated to performance. For example, if the guest is a service provider without any traffic, it may value any number of cache ways as contributing zero to its utility. We denote the guest's valuation by

$$V(cache, state) = V_{perf}(perf(cache, state)) ,$$

where $V_{perf}(perf)$ describes the value derived by the client for a given level of performance for a given guest, and $perf(cache, state)$ describes the performance the guest can achieve given its current state and a certain number of cache ways. $V_{perf}(perf)$ is private for each client; it is based on economic considerations and business logic.

For example, two clients run a market forecasting algorithm and need to evaluate 1,000 stocks on average to find a group of stocks that are expected to yield 10% profit. They can measure their performance in evaluated stocks per hour. The first client is willing to invest \$10K. For this client, $V_{perf}(perf) = \frac{\$1}{\text{stock}} \cdot perf$. The second client, however, is only willing to invest \$1K. For this client, $V_{perf}(perf) = \frac{\$0.1}{\text{stock}} \cdot perf$. Both clients will need to know $perf(cache, state)$: how many stocks they can evaluate per hour when given various numbers of cache ways and under the current conditions (e.g., server load).

In our experiments, we use an offline mapping of performance as a function of cache and the current server load. We found this to be sufficiently accurate, as we demonstrate in §8.2. But performance can also be measured online, as demonstrated in a number of works [6, 19, 41, 58, 64, 66], and as might be required in real-world scenarios.

5 Auction Protocol

In *Ginseng*, each client pays a constant hourly fee for its guest VM while it is assigned to the *pit*. In each auction round, each guest can bid for exclusive cache ways. After each round, *Ginseng* calculates a new cache allocation,

and guests exclusively rent the cache ways they won until the next round ends.

The constant fee is not affected by the auction results. It guarantees the lion's share of the host's revenues, so that the host can utilize the auction to maximize social welfare, thereby attracting more guests.

Clients with hard performance requirements can verify the availability of exclusive cache ways by prepaying for them (and thereby removing them from the cache ways that are up for rent). Supporting these clients will be easier in future hardware with more COSes. These clients are not included in our experiments, which were performed on Haswell. Clients with very low performance requirements are expected to pay in advance only the constant fee and bid with low valuations or not at all, so that they rarely pay for cache ways and manage to stay within their budget. Clients in between those extremes are expected to choose a flexible payment scheme that meets their needs.

Here follows the description of an auction round, along with a numeric example. In the example, as well as in the experiments that follow, the host's clients are service providers with their own customers.

Initialization. Each guest is assigned to the *pit* as it enters the system.

Auction Announcement. The host informs each guest of the number of available cache ways, the server load (i.e., the number of active VMs) and the auction's closing time, after which bids are not accepted. In our example, the physical machine has 20 cache ways, two of which are dedicated to the *pit*, so the host announces an auction for 18 cache ways.

Bidding. Interested guests bid for cache ways. A bid is composed of a price per hour for each number of exclusive cache ways that the guest is willing to rent. In our example, 10 guests choose not to bid in this round, and 2 guests have strict performance requirements: Guest 1 is willing to pay \$1 per hour when allocated 10 or more cache ways and \$0 per hour for fewer cache ways. Guest 2 is willing to pay \$5 per hour for allocation of 14 or more cache ways and \$0 per hour for fewer cache ways.

Bid Collection. The host asynchronously collects guest bids as soon as the auction is announced. It considers the most recent bid from each guest, dismissing earlier bids. Guests that do not bid lose the auction automatically, and are assigned to the *pit*.

Allocation and Payments. The host computes the allocation and payments according to the VCG auction rules, using a specially designed algorithm described in §6. For each guest, it computes how much cache it won and at what price. The payment rule guarantees that the guest will not pay a price that exceeds its bid. The guest's account is charged accordingly (and accurately, by the second). In the example, guest 1 loses, is assigned to the *pit* and pays nothing; guest 2 wins all of the cache ways and pays \$1 per hour.

Informing Guests and Assigning Cache Ways. The host informs each guest of the auction results that are relevant to it: its cache allocation and payment. Then, the host takes cache ways from those who lost them and gives them to those who won, by updating their COSes as necessary.

6 Auction Rules

Every auction has an allocation rule—who gets the goods?—and a payment rule—how much do they pay? To determine who gets the goods, the VCG algorithm calculates the optimal allocation of cache ways: the one that maximizes social welfare—client satisfaction—as described in §4. To determine the optimal allocation, the VCG auction solves a constrained multi-unit allocation problem, as detailed in §6.1. To determine how each client pays, the VCG auction computes the damage it inflicts on other guests, as detailed in §6.2. After explaining the auction rules, we discuss their run-time complexity and provide an example showing how they are executed. The correctness proof can be found in the full version [13].

6.1 Allocation Rule

To find the optimal allocation—the one that maximizes the social welfare—*Ginseng* must consider all the allocations for the number of guests, the number of cache ways available, the size of the *pit*, and the maximum number of classes-of-service (COS) available. Since the number of possible allocations is exponential in the number of guests and cache ways, iterating over them is impractical. Therefore, we introduce a simple algorithm that finds the optimal allocation in polynomial time.

First, the algorithm combines two guests into an effective guest with a joint valuation function. For any number of cache ways that the two guests will get, the joint function stores the optimal division of cache ways between the two guests, and returns the sum of the valuations of these guests for that cache way division. Then, in each step, it continues to combine the guests and the effective guests until a single effective guest remains. Its valuation function returns the maximal aggregated valuation of all the guests, which is the social welfare. The optimal allocation is then reconstructed from stored division data of the joint valuation functions.

6.2 Payment Rule

The payments follow the VCG exclusion compensation principle, as formulated in [4]. Let a_k denote player's k cache allocation, and let a'_k denote the number of cache ways that would have been allocated to guest k

in an auction in which guest i did not participate and the rest of the guests bid as they bid in the current auction. Then guest i is charged a price p_i, computed as follows:

$$p_i = \sum_{k \neq i} V_k(a'_k) - V_k(a_k) \ .$$

The payment reflects the damage that guest i's bid inflicted on other guests.

6.3 Complexity

Let N denote the number of bidding guests. Let W denote the total number of cache ways and let C denote the total number of COSes.

$V_{combined}(w,c)$ is obtained by comparing $O(w \cdot c)$ allocations and summing the two valuations for each allocation. That is, for $W \cdot C$ values, the time complexity is $O\left(W^2C^2\right)$. After $N-1$ reductions we will have one combined valuation. So the total time complexity of the allocation algorithm is $O\left(W^2 \cdot C^2 \cdot N\right)$.

To compute the payment for a guest that is allocated any cache ways, the allocation algorithm needs to be computed again without this guest. Since the number of winning guests is bounded by C, in each auction round the allocation procedure is called up to $\min(C,N)+1$ times, and the time complexity of the total allocation and payment calculation is $O\left(W^2 \cdot C^2 \cdot N \cdot \min(C,N)\right)$.

The algorithm runtime was reasonable: less than one second using a single hardware thread, even when tested with thousands of cache ways and guests, and an unlimited number of COSes, in preparation for future architectures.

7 Experimental Setup

In this section we describe the experimental setup in which we evaluate *Ginseng*.

7.1 Machine Setup

We used a machine with two Intel(R) Xeon(R) E5-2658 v3 @ 2.20GHz CPUs with a 30MB, 20-way LLC that supports CAT. Each CPU had 12 cores with hyper-threading enabled, for a total of 48 hardware threads. One CPU was dedicated to the host and the other to the guests. As many guests as possible were each pinned to two exclusive hardware threads that resided on the same core. In experiments with more than 12 guests, some were pinned to one hardware thread each. This let us manage cache allocation per hardware thread and not per VM process. The machine had 32GB of RAM per socket. Each VM got 1GB of RAM, pinned to memory from the same node. The host ran Ubuntu Linux with kernel 4.0.9-040009-generic #201507212131, and the guests ran 3.2.0-29-generic-#46-Ubuntu.

Each application ran exclusively on a virtual machine (VM); hence we refer from now on to an application and the guest VM running it interchangeably. However, this is not compulsory; in real scenarios, the VM's valuation can change in each bid to cater to changing conditions or changing applications, as is customary in the cattle model of cloud computing.

7.2 Workloads

The Phoronix Test Suite [37] includes over 100 benchmarks for a variety of applications. We chose a sample of 10 applications with varying cache utilization, along with their associated benchmarks: ***BZIP2*** (1.5.0) uses parallel compression on a 256MB file. ***H.264*** (2.0.0) encodes a video to H.264 format on the CPU. ***HMMer*** (1.1.0) searches the Pfam database for profile hidden Markov models. ***Gcrypt*** (1.0.3) uses the CAMELLIA256-ECB cipher. ***OpenSSL*** (1.9.0) uses an open-source SSL implementation with 4096-bit RSA. Five of the applications were taken from the ***SciMark 2.0 suite*** [51] (1.2.0), which is included in the Phoronix suite but exists also as a stand-alone: ***Fast Fourier Transform*** performs a one-dimensional forward transform of complex numbers. ***Dense LU Matrix Factorization*** computes the LU factorization of a dense matrix using partial pivoting. ***Monte-Carlo*** approximates the value of pi by using random point selection on a circle. ***Jacobi Successive Over-Relaxation*** performs Jacobi successive over-relaxation on a grid. ***Composite-Scimark*** is comprised of several SciMark 2.0 benchmarks. The following subsection shows the performance measurements of these benchmarks.

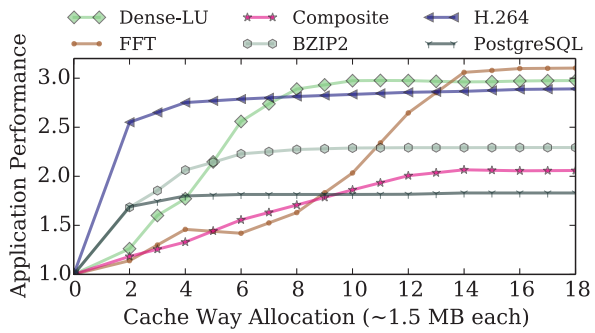
We also tested some larger, commonly used applications such as ***PostgreSQL*** and ***Memcached***. However, we eventually decided not to use them in the experimental section as both require long warm-up periods and would reduce the number of experiments we were able to perform. The performance measurements of both these applications are also shown in the following subsection. We used the TPC-B benchmark with 10 clients to test *PostgreSQL*, which ran on a VM with 4GM of RAM. To test *Memcached* we used memslap with 64-byte values and 90% reads, and configured *Memcached* to use 64MB of RAM.

7.2.1 Classifying the Applications

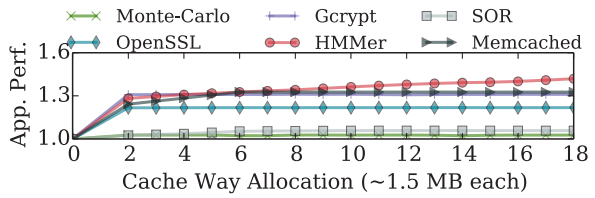
We used the benchmarks to classify the above applications and demonstrate how they perform under different cache allotments and partitioning.

Cache-utilizer applications perform better when allocated more cache. The performance of such applications is depicted in Figure 2a.

Cache-neutral applications cannot utilize the cache to

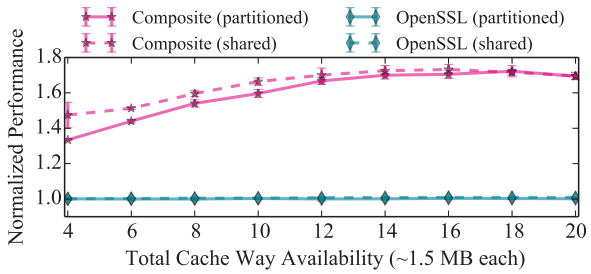


(a) Cache-utilizer application performance increases with more ways.

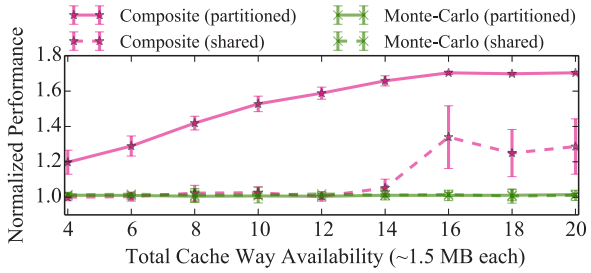


(b) Cache-neutral application performance is indifferent to cache ways.

Figure 2: Performance of various applications, normalized by their performance in the *pit*. Measured with 11 other guests assigned to the *pit*. All of the applications were allocated two hardware threads.



(a) *Composite-Scimark* (cache-utilizer) performance improves when sharing cache with *OpenSSL* (cache-neutral). Therefore, we consider *OpenSSL* to be a non-polluter.



(b) *Composite-Scimark* (cache-utilizer) performance drops when sharing cache with *Monte-Carlo* (cache-neutral, does not gain from extra cache). Therefore, we consider *Monte-Carlo* to be a cache-polluter.

Figure 3: *Composite-Scimark* performance when sharing the cache with cache-polluter vs. non-cache-polluter applications. The performance was normalized to the minimum measurement in all the experiments.

obtain better performance. The performance of such applications is depicted in Figure 2b. However, they might experience minor improvement as compared to being assigned to the *pit*.

Cache-polluter applications are cache-neutral applications that pollute the cache in a way that will harm the cache-utilizer's performance when cache is shared with the polluter. To demonstrate this, we ran several experiments, in each of which we ran one cache-utilizer and 7 cache-neutral applications simultaneously. We assigned all of the cache to all of the guests in the shared scenario. In the partitioned scenario, we assigned 2 cache ways to all of the cache-neutral applications together and the rest of the available cache was allocated to the cache-utilizer. Figure 3 shows that the cache-utilizer's performance drops when sharing cache with a cache-polluter application.

It is likely that partitioning the cache can benefit cache-utilizer applications by protecting them from cache polluters without affecting cache-neutral applications. Furthermore, the provider may need to decide how to allocate the cache between several cache utilizers.

7.2.2 Living with Offline Profiling

Offline profiling is error-prone due to the dynamic nature of the cloud. For example, a cache-utilizer may depend on memory bandwidth. That is, if an application can benefit from faster access to the memory via cache, it will likely suffer when memory access time increases due to low memory bandwidth. Memory bandwidth isolation mechanisms have been researched [27, 29, 46, 48], but are not yet available in commercial hardware [41]. Thus, we are compelled to accept the available memory bandwidth as dependent on the number of guests in the cloud. In a real cloud, the client might want to receive information from the host about its available (or expected) memory bandwidth and take it into account when deriving its valuation. In our *Ginseng* experiments, we consider the number of guests in the system to be the only factor influencing memory bandwidth and report it to each guest. The guest uses this information from the host as a factor in its valuation, employing its offline performance profiling for environments with various numbers of guests (Figure 4).

7.2.3 Valuations

The experimental scenario consists of cloud guests who are themselves service providers. Each guest serves one

of its customers at a time. Each guest's customer shares performance metrics but has different performance requirements. Thus, when customers change, this implies a change in the guest's valuation function. The valuation function is formulated as the profiled performance function, normalized to the range $[0..1]$, and multiplied by a scale factor that represents the amount the guest's customer is willing to pay for the performance. The scale factor depends on the performance: if it is below the customer's required performance, then the scale factor will be lower. Formally, we can express this as:

$$valuation(a) = s(perf(a)) \cdot \frac{perf(a) - min\_perf}{max\_perf - min\_perf},$$

where a denotes the cache way allocation and s denotes the scale factor. The *pit* is free of charge, and therefore $valuation(0) = 0$.

We characterize three customer types by their scale factors: A ***low-valuation customer*** has a constant scale factor $s = 0.05$. Such a client is unconcerned with performance or unwilling to pay to improve it. An ***medium-valuation customer*** has a scale factor $s = 1$ when meeting its performance requirements, and $s = 0.05$ otherwise. A ***high-valuation customer*** has a scale factor $s = 3$ when meeting its performance requirements, and $s = 0.05$ otherwise. The performance requirements of this type of customer are higher. See, for example, the valuation functions of a customer running *Composite-Scimark* (Figure 5).

In each experiment, each guest serves 10 customers with different valuations, one after the other. We emulated that by giving each guest a pool of valuations with four customer-type distributions. The distributions are denoted as triplets of high-valuation, medium-valuation and low-valuation customers. We experimented with the following distributions: (**1,1,8**), (**1,2,7**), (**0,5,5**), and (**3,3,4**). For each guest we employed a different, randomly shuffled and unique order on those valuation sets. Hence, when we repeated an experiment but with more guests, a guest that participated in both experiments had the same valuation order in both. This gives us an idea of what we could achieve if we consolidate more guests on the same physical host.

7.3 Alternative Cache Allocation Methods

We compared *Ginseng* with the following cache allocation methods:

Shared-cache allocation, where all of the guests share the entire LLC. This was the prevalent method prior to the introduction of CAT.

Uniform-static allocation, where each guest is allocated a fixed and equal number of cache ways, as many as the hardware allows. In our hardware there are 4 COSes, so for 4 clients or fewer the cache was divided equally.

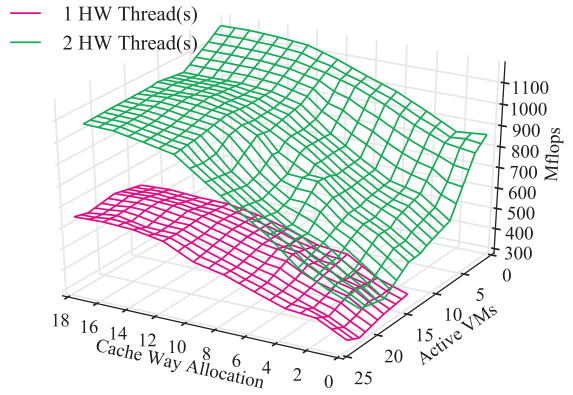
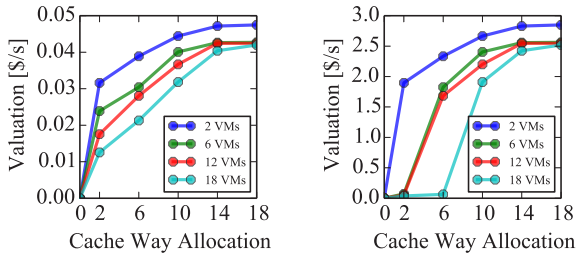


Figure 4: Example of performance profiling: *Composite-Scimark* under different server loads (i.e., active VMs) and with different numbers of allocated hardware threads.



(a) Low-valuation customer (b) High-valuation customer

Figure 5: *Composite-Scimark* valuation function for different server-loads (i.e., active VMs) and when allocated 2 hardware threads. Note the different scale in the vertical axes.

For more clients, three clients received six cache ways each, and the rest of the clients were assigned to the *pit*.

Performance-maximizing allocation, where the guests' allocation maximizes the overall performance of all of the applications. To this end, we employed *Ginseng*'s optimization algorithm to maximize the aggregate performance by using a constant scale factor $s = 1$ for all the guests' valuations. We did not compare to this method when the experiment had more than one type of application, as the aggregated performance of different applications is meaningless. This allocation is in practice a static allocation, as there is no provider-observable difference in the application's behavior during the experiment.

Ideal-static allocation, where all the future client valuations are known in advance, and the static allocation that maximizes the social welfare is chosen. It serves as an upper bound for all the static allocations.

7.4 Time Scales

Ginseng's responsiveness to guest valuation changes improves with more frequent auctions. Hence, an auction round is conducted every 10 seconds. In each round, the host collects guest bids for 3 seconds, and computes the optimal allocation and payments for at most 3 seconds (in practice it takes well under one second). Then the host notifies the guests of their new allocation and payments and applies the new allocation. However, to gather enough performance measurements for our experiments, we changed the guest's valuation every 5 minutes in the dynamic allocation experiments. In the static allocation experiments, where valuation changes did not affect the guests' state, 30 seconds were enough.

8 Evaluation

Our experiments were designed to answer the following questions: (1) Which cache allocation method results in guests who are most satisfied (i.e., have the highest social welfare)? (2) How accurate is off-line profiling of guest performance? (3) What are the limitations of a *Ginseng*-based cloud?

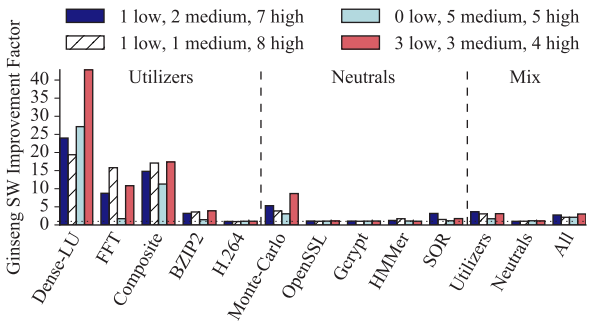
The data presented in this paper is based on 4,287 experiments, each lasting 10-50 minutes.

8.1 Comparing Social Welfare

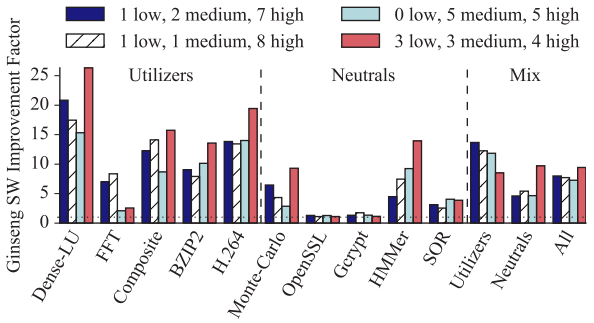
We evaluated the social welfare achieved by Ginseng vs. each of the four other methods listed in §7.3, for all of the workloads and for workload mixtures (neutrals, utilizers, and a mixture of both). We varied the number of guests running the relevant applications. In the mixed workload experiments we cyclically chose the new workload from the set.

The social welfare was calculated from the measured performance of each application using its guest's valuation function. *Ginseng* achieves much better social welfare than the other allocation methods for the tested workloads, as seen in Figure 6. It improves social welfare for *Dense LU Matrix Factorization* by up to 42.8× compared to shared-cache and by up to 26.3× compared to ideal-static. For *Fast Fourier Transform* and *Composite-Scimark*, *Ginseng* improves social welfare by 1.7× to 17.1× compared to shared-cache and ideal-static. For a heterogeneous cloud with cache-utilizers, *Ginseng* improves social welfare by up to 13.7× compared to other allocation methods.

As seen in Figures 7a,7b, *Ginseng* increases the social welfare for an increasing number of up to 12 guests, because more high-valuation and medium-valuation customers can be served simultaneously. However, other methods, including performance-maximizing, improve the social welfare very little or not at all with more guests because they disregard client valuation changes.



(a) *Ginseng* improvement factor over shared-cache allocation method.



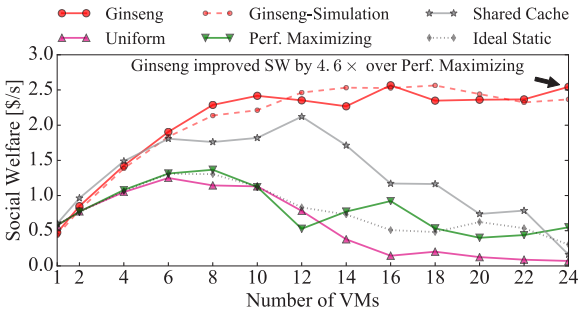
(b) *Ginseng* improvement factor over ideal-static allocation method.

Figure 6: Maximum improvement factor of *Ginseng* compared to the shared-cache and ideal-static methods with different assumptions on the number of high, medium, and low valuation customers. The maximum is over any number of guests with the application, or mixture of applications.

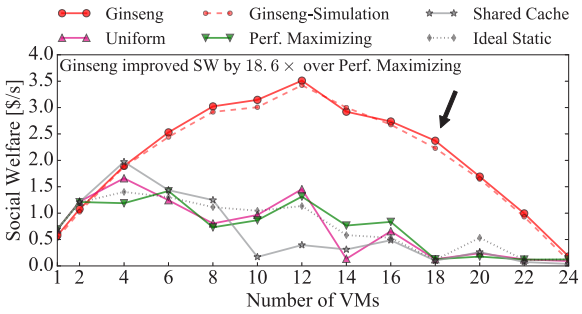
For more than 12 guests, hardware threads become a bottleneck, and some guests only get one hardware thread; hence the social welfare gradually declines (Figure 7b). However, under *Ginseng*, some applications can compensate for fewer hardware threads with additional ways, so that *Ginseng* can maintain high social welfare while increasing server consolidation (Figure 7a).

Nevertheless, other allocation methods can still produce results closer to *Ginseng* for some specific scenarios. For example, when all guests run cache-neutral applications (Figure 7c), the applications are less likely to suffer from being consigned to the *pit* than when some guests run cache-utilizer applications. Although their performance does not depend strongly on cache allocation, their performance in the *pit* deteriorates when more guests are assigned to it. Thus, as the number of guests in the cloud increases, it becomes increasingly important to allocate cache ways to the right guests, as opposed to assigning them to the *pit*.

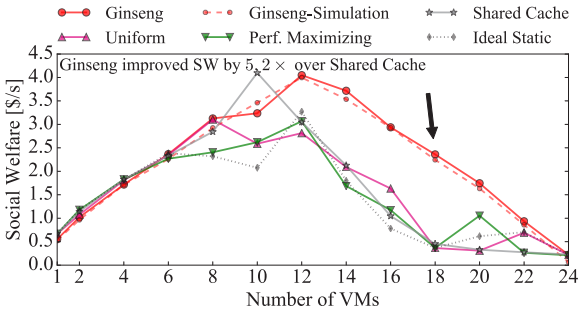
Shared-cache can produce better results than *Ginseng* when all guests use applications with a small memory



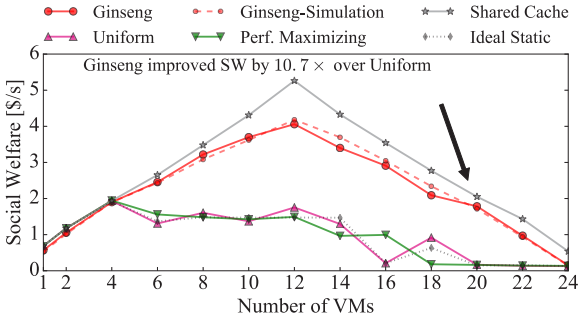
(a) All guests run *Fast Fourier Transform* with 1 high-valuation customer, 1 medium-valuation customer and 8 low-valuation customers. *Ginseng* improves social welfare by up to 4.7× over performance-maximizing and by up to 15.8× over shared-cache.



(b) All guests run *Dense LU Matrix Factorization* with 1 high-valuation customer, 2 medium-valuation customers and 7 low-valuation customers. *Ginseng* improves social welfare by up to 18.6× over performance-maximizing and by up to 24× over shared-cache.



(c) All guests run *Monte-Carlo* with 1 high-valuation customer, 2 medium-valuation customers and 7 low-valuation customers. *Ginseng* outperforms other allocation methods as server consolidation is increased, even for cache-neutral applications.



(d) All guests run *H.264* with 1 high-valuation customer, 2 medium-valuation customers and 7 low-valuation customers. This is the only case where shared-cache outperforms *Ginseng* for any number of guests.

Figure 7: Social welfare under different cache allocation methods as a function of the number of guests. The dashed lines indicate an experiment where the clients perform identically to the profiler (artificial clients). Cache-utilizer applications can greatly benefit from *Ginseng*. Cache-neutral applications can still enjoy the benefits of *Ginseng*, albeit to a lesser extent. Applications with a small memory working set will prefer sharing the cache with others like it.

working-set (Figure 7d). In such a case, cache misses are rare (e.g., a solid 80% hit ratio for 12 *H.264* with shared-cache). Thus, because none of the applications access the memory frequently, an application is expected to consume the maximum memory bandwidth when it does. Hence, memory bandwidth will not be a bottleneck in this case. However, when memory bandwidth is low due to frequent memory access by other applications, even a rare memory access can dramatically affect performance. This is illustrated in Figure 9a, where two applications are *H.264* and 10 applications are *Monte-Carlo*. *H.264* uses a small memory working-set but relies on prefetching to improve performance. When the cache is not shared, all the prefetched data remains in the cache, resulting in better performance. When the two *H.264* applications share the cache, and the 10 *Monte-Carlo* applications are assigned to the *pit*, some of the *H.264* data might be evicted from the cache. Because the *Monte-Carlo* applications access the memory very frequently, any memory access by the *H.264* applications will result in sharply decreased performance of the latter.

Although our primary concern is improving the social welfare, it is interesting to monitor the commonly considered metric of aggregated performance. This metric is only applicable in the scenario where all the guests run the same application. In these experiments we conducted, the aggregated performance improves slightly with *Ginseng* in most cases. In some cases, the shared-cache or performance-maximizing methods improve the aggregated performance by up to 10% in comparison to *Ginseng*. The only exception is *H.264*, which in some cases yielded a 200% improvement in the aggregated performance with shared-cache, due to, as we mentioned above, its small memory working-set. This does not di-

minish the above because the applications are not considered equal, in contrast to what standard performance improvement methods assume.

8.2 Influence of Off-Line Profiling

We experimented with off-line performance profiling data (e.g., Figure 4) that was measured in a controlled environment. However, in a live environment, profiling data should be collected on-line, so that it remains fresh under changing conditions. To retrospectively justify the use of off-line profiling in our experiments, we measured the deviation of actual performance from performance predicted by the off-line profiling (for the conditions at the time).

In Figure 8, we see that the deviation from the expected performance was under 10% in most cases. Moreover, the median deviation for all the applications was under 1%, and 95% of the measurements deviate from the predicated performance by less than 12%. The accuracy of the profiling is reflected in the small difference between *Ginseng* and the simulation (Figure 7), and shows that a more accurate profiler would achieve only a minor improvement.

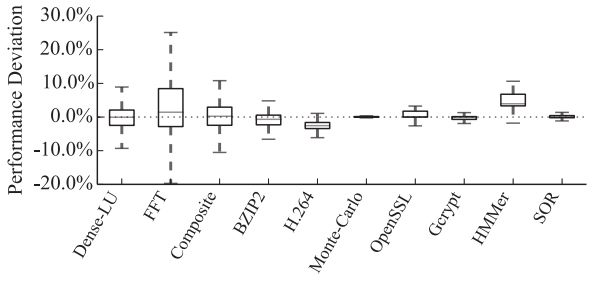


Figure 8: Expected performance deviation for all applications in all of our experiments. The *pit* measurements are excluded as the performance is expected to fluctuate when sharing a small number of cache ways.

8.3 To Share or Not To Share

We have already seen cases where partitioning the cache can benefit cache-utilizer applications without affecting cache-neutral ones. However, in some cases, a partitioning that includes limited sharing could greatly improve overall performance.

We consider two possible simple partitioning schemes where we reserve two cache ways for the *pit*. ***Hard-partitioning*** allocates a set of exclusive cache ways to each guest. A guest that values cache more than others will be allocated more cache ways. ***Soft-partitioning*** allocates all the cache ways to the guest that values cache the most. The guest that values cache second-most gets a subset of the previous guest's ways, and so forth. For simplicity, we only let guests bid for the right to use fixed COSes (for example: $COS_1 = [1..2]$ (pit), $COS_2 = [3..20]$, $COS_3 = [3..15]$, $COS_4 = [3..10]$). Guests will need to consider how they value these COSes, assuming other COSes may be occupied by at most one application per COS.

As we have seen, guests can successfully estimate their expected performance for a given allocation of exclusive cache (i.e., hard-partitioning). However, it is harder to valuate a given soft-partitioning allocation when the cache is shared with an unknown guest, as is common in the cloud. Even if the neighbor guest is known, the performance and valuation still depend on additional dimensions (quantity and share level) that further complicate the bidding and optimization process for guest and host alike.

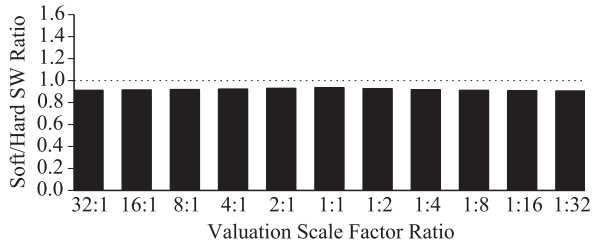
Ginseng uses hard-partitioning due to its simplicity and accuracy of estimation. In this section we assess the benefit guests might have achieved from soft-partitioning. To simplify, we tested several pairs of cache-utilizer applications, and the *pit* contained 10 *Monte-Carlo* applications that served as cache-polluters. We measured the performance of each pair for all possible cache allocations in the hard-partitioning and soft-partitioning allocation schemes. Then, we compared each pair's performance in these settings. We used each application's measured performance, normalized to its performance when assigned to the *pit*, as its valuation function, and experimented with different ratios of scale factor between each pair's valuations.

Although soft-partitioning sometimes yields better social welfare than hard partitioning (Figure 9b), it usually improves it by no more than 10% (Figures 10 and 9a), or even degrades it.

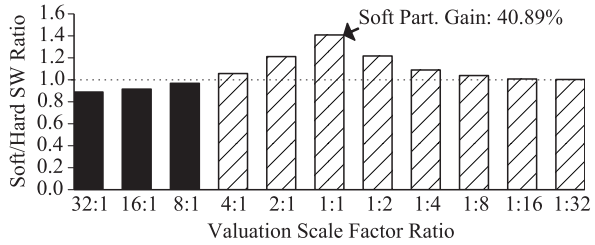
8.4 Dynamic Allocation Overhead

Transferred cache ways require a warm-up period. Moreover, they are likely to contain the previous application's data. If the previous application is a cache-utilizer, it is likely to access this data soon, and have this data marked as most-recently-used (MRU). This creates competition for the other application. If it accesses its own data too slowly, it may end up evicting that data from its previously owned ways to store new data in the cache. It will thus take longer (possibly forever) for the second application to benefit from additional cache ways. We refer to such a scenario as *cache leakage*.

Furthermore, any allocation change is constrained by the need to preserve the consecutiveness of ways in a COS. For example, let the initial allocation be $COS_1 = [1..4]$, $COS_2 = [5..6]$ and $COS_3 = [7..10]$. To transfer a way from COS_3 to COS_1, COS_2 must also change. The least disruptive transfer moves two ways: way 7 to COS_2 and way 5 to COS_1. Compared with the required transfer



(a) Both applications are *H.264*. Hard-partitioning yields better social welfare than soft-partitioning for all ratios.



(b) Both applications are *Fast Fourier Transform*. The maximal improvement for soft-partitioning over hard-partitioning is achieved when the applications' scale factors are equal.

Figure 9: Social welfare under hard vs. soft-partitioning. Striped columns indicate better social welfare under soft-partitioning. Black indicates the opposite.

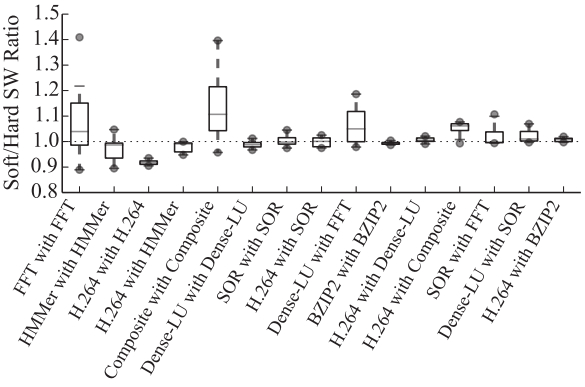


Figure 10: Social welfare improvement under soft-partitioning compared with hard-partitioning for various application pairs. Boxes show the middle 50% of the values over different valuation scale factor ratios. Whiskers mark extreme values.

of a single way, this consecutiveness-constrained transfer doubles the *cache leakage* effect.

We measured how dynamic allocation changes affect application performance. In each experiment, a guest machine ran one of the workloads listed in §7.2. At the same time, the host natively ran an application that repeatedly touches all its data, in parallel, using 8 hardware threads and by utilizing the CPU's out-of-order-execution (OOOE) mechanism. We designed this application to ensure that its data fits perfectly in its allocated cache ways, by detecting cache lines that reside on the same cache set [24, 63]. When an application keeps its cache lines marked as MRU, the *cache leakage* effect is amplified, and thus represents a worst-case scenario.

Each experiment ran for 10 minutes. In each experiment both applications were allocated a basic set of ways. Another set was transferred between the applications every $[10..60]$ seconds. The numbers of basic and transferred ways were in the range $[2..10]$.

In the baseline experiments the cache ways were transferred once, from the application, to ensure that the application's performance was not affected by the *cache leakage*. Half the performance measurements in these experiments were high and half were low.

In the experiments with the frequent transfer intervals, there is a similar performance distribution, whose values varied by up to 4% from the baseline values (high values were lower, low values were higher). The mean performance over the duration of the experiment varied from the baseline by up to 1.1% for all of the workloads. Mean performance values did not depend on the transfer frequency: the effect of a single transfer is negligible, and when there are many intermittent leaks, those that benefit an application will compensate for those that harm it.

9 Related Work

Market Driven Resource Allocation. Lazar and Semret auctioned bandwidth [38]. Agmon Ben-Yehuda et al. introduced Ginseng as a memory auctioning platform [4]. Drexler and Miller [11] and Maillé and Tuffin [44] suggested an auction to compute a market clearing price for memory and bandwidth, respectively. Waldspurger et al. auctioned processor time slices [61].

Cache Partitioning [56]. Many hardware solutions detect cache pollution by non-reused data and prevent its future insertion, or apply partitioning to prevent the application from interfering with other applications [12, 14, 28, 31, 40, 50, 52]; others rely on the user or OS to allocate the cache, like CAT does [9, 33, 39, 55]. However, CAT is the first hardware implementation of such a mechanism in commodity hardware.

A cache-polluter can prevent caching of specific data by using Intel's non-temporal store instruction. Cache can be partitioned in software using page-coloring [59] to prevent cache pollution: by the program [8, 58], by the OS [19, 42, 65, 66], and under virtualized environments [30, 54, 62]. Some works proposed to guarantee the applications' performance demands via LLC management [18, 20, 21, 41, 47, 53, 57]; these works require the

guests to reveal their performance requirements without any incentive to do so.

Although page-coloring allows a finer granularity in the cache allocation, it will not be as effective as CAT for this work as it requires that memory be moved in order to change the cache allocation, which will place a heavy burden on both the clients and the provider.

Shared Cache Performance Interference. VM *Performance interference* when sharing LLC [16,34,36] was analyzed and predicted. Such methods can help guests estimate their performance on shared cache and allow a biddable soft-partitioning scheme.

10 Conclusions and Future Work

Ginseng efficiently allocates cache to selfish black-box guests while maximizing their aggregate benefit. *Ginseng* can also benefit private clouds, where it distinguishes between guests that perform the same function for different purposes, such as a test server vs. a production server. Cache-*Ginseng* is the first economically-based cache allocation method, and cache is the second resource implemented in the *Ginseng* framework. Cache-*Ginseng* works by hard-partitioning the cache in short intervals according to a VCG auction in which the guests have an incentive to bid their true valuation of the cache.

The guests utilize their cache fast enough to allow such rapid changes in the allocation without any substantial effect on their performance. *Ginseng* achieves up to 42.8× improvement in social welfare when compared with alternative cache allocation methods. Shared cache allocation may improve on these results. Formulating a bidding and valuation language for shared cache remains as future work.

Although the VCG auction has a high computational complexity, the coarse cache allocation granularity makes it suitable for cache auction. Similarly, it can be efficiently used to allocate other small numbered multi-unit resources whose valuation functions are monotonically rising: CPUs, for example. Thus, *Ginseng* is not only a platform for auctioning cache and memory, but also a concrete step toward the Resource-as-a-Service (RaaS) cloud [1, 3], in which all resources, not just cache and memory, will be bought and sold on-the-fly. Extending *Ginseng* to additional resources and to their concurrent allocation remains as future work.

For *Ginseng* to be applicable in real public or private clouds, further work is required to create tools for clients to evaluate their expected performance with different resource allocations, where the parameters of the cloud are dynamic (e.g., online profiling), and to assist the clients in valuating their performance in economic terms. Furthermore, to maximize the social welfare over an entire cloud to prevent overcrowded machines, VM migration support should be implemented in *Ginseng* in a way that takes into account the economic benefit and cost to the client and the provider.

11 Acknowledgments

We thank Sharon Kessler, Nadav Amit, Muli Ben-Yehuda, Moshe Gabel, Avi Mendelson, Vikas Shivappa, Priya Autee, Edwin Verplanke and Matt Fleming for fruitful discussions. This work was partially funded by the Hasso Platner Institute, by the Professor A. Pazi Joint Research Foundation and by the Israeli Ministry of Science. We thank Intel for loaning the hardware that facilitated the research.

References

[1] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The resource-as-a-service (RaaS) cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud)* (2012), USENIX Association.

[2] AGMON BEN YEHUDA, O., BEN YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation (TEAC) 1*, 3 (2013).

[3] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The rise of RaaS: The resource-as-a-service cloud. *Communications of the ACM 57*, 7 (2014), 76–84.

[4] AGMON BEN-YEHUDA, O., POSENER, E., BEN-YEHUDA, M., SCHUSTER, A., AND MU'ALEM, A. Ginseng: Market-driven memory allocation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2014), ACM, pp. 41–52.

[5] AGMON BEN-YEHUDA, O., SCHUSTER, A., SHAROV, A., SILBERSTEIN, M., AND IOSUP, A. ExPERT: Pareto-efficient task replication on grids and a cloud. In *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)* (2012), IEEE, pp. 167–178.

[6] ALBONESI, D. H. Selective cache ways: on-demand cache resource allocation. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32)* (1999), IEEE, pp. 248–259.

[7] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., AND ZAHARIA, M. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[8] BUGNION, E., ANDERSON, J. M., MOWRY, T. C., ROSENBLUM, M., AND LAM, M. S. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1996), ACM, pp. 244–255.

[9] CHIOU, D., JAIN, P., RUDOLPH, L., AND DEVADAS, S. Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Annual Design Automation Conference (DAC)* (2000), ACM, pp. 416–419.

[10] CLARKE, E. Multipart pricing of public goods. *Public Choice 11*, 1 (1971), 17–33.

[11] DREXLER, K. E., AND MILLER, M. S. Incentive engineering for computational resource management. *The Ecology of Computation 2* (1988), 231–266.

[12] DYBDAHL, H., AND STENSTRÖM, P. Enhancing last-level cache performance by block bypassing and early miss determination. In *Advances in Computer Systems Architecture*, C. Jesshope and C. Egan, Eds., vol. 4186 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 52–66.

[13] FUNARO, L., AGMON BEN-YEHUDA, O., AND SCHUSTER, A. Ginseng: Market-driven LLC allocation. Tech. rep., Technion—Israel Institute of Technology, 2016. http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-info.cgi/2016/CS/CS-2016-03.

[14] GONZÁLEZ, A., ALIAGAS, C., AND VALERO, M. A data cache with multiple caching strategies tuned to different types of locality. In *ACM International Conference on Supercomputing 25th Anniversary Volume* (2014), ACM, pp. 217–226.

[15] GORDON, A., HINES, M., DA SILVA, D., BEN-YEHUDA, M., SILVA, M., AND LIZARRAGA, G. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. In *3rd IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2011).

[16] GOVINDAN, S., LIU, J., KANSAL, A., AND SIVASUBRAMANIAM, A. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC)* (2011), ACM.

[17] GROVES, T. Incentives in teams. *Econometrica: Journal of the Econometric Society* (1973), 617–631.

[18] GUO, F., SOLIHIN, Y., ZHAO, L., AND IYER, R. Quality of service shared cache management in chip multiprocessor architecture. *ACM Transactions on Architecture and Code Optimization (TACO) 7*, 3 (2010).

[19] GUO, R., LIAO, X., JIN, H., YUE, J., AND TAN, G. NightWatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *Proceedings of the USENIX Technical Annual Conference* (2015), USENIX Association, pp. 307–318.

[20] GUPTA, D., CHERKASOVA, L., GARDNER, R., AND VAHDAT, A. Enforcing performance isolation across virtual machines in Xen. In *Middleware* (2006), M. van Steen and M. Henning, Eds., vol. 4290 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 342–362.

[21] HASENPLAUGH, W., AHUJA, P. S., JALEEL, A., STEELY, S., AND EMER, J. The gradient-based cache partitioning algorithm. *ACM Transactions on Architecture and Code Optimization (TACO) 8*, 4 (2012).

[22] HEO, J., ZHU, X., PADALA, P., AND WANG, Z. Memory overbooking and dynamic control of Xen virtual machines in consolidated environments. In *IFIP/IEEE International Symposium on Integrated Network Management (IM)* (2009), IEEE, pp. 630–637.

[23] HINES, M. R., GORDON, A., SILVA, M., DA SILVA, D., RYU, K., AND BEN-YEHUDA, M. Applications know best: Performance-driven memory overcommit with Ginkgo. In *2011 IEEE 3rd International Conference on Cloud Computing Technology and Science (CloudCom)* (2011), IEEE, pp. 130–137.

[24] HUND, R., WILLEMS, C., AND HOLZ, T. Practical timing side channel attacks against kernel space ASLR. In *IEEE Symposium on Security and Privacy (SP)* (2013), IEEE, pp. 191–205.

[25] INTEL. Improving real-time performance by utilizing cache allocation technology. http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html, 2015. Accessed May, 2016.

[26] INTEL OPEN SOURCE.ORG. Cache monitoring technology, memory bandwidth monitoring, cache allocation technology & code and data prioritization. https://01.org/packet-processing/cache-monitoring-technology-memory-bandwidth-monitoring-cache-allocation-technology-code-and-data. Accessed January, 2016.

[27] IYER, R., ZHAO, L., GUO, F., ILLIKKAL, R., MAKINENI, S., NEWELL, D., SOLIHIN, Y., HSU, L., AND REINHARDT, S. QoS policies and architecture for cache/memory in CMP platforms. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2007), vol. 35, ACM, pp. 25–36.

[28] JAIN, P., DEVADAS, S., AND RUDOLPH, L. Controlling cache pollution in prefetching with software-assisted cache replacement. *Comptation Structures Group, Laboratory for Computer Science CSG Memo 462* (2001).

[29] JEONG, M. K., EREZ, M., SUDANTHI, C., AND PAVER, N. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proceedings of the 49th Annual Design Automation Conference (DAC)* (2012), ACM, pp. 850–855.

[30] JIN, X., CHEN, H., WANG, X., WANG, Z., WEN, X., LUO, Y., AND LI, X. A simple cache partitioning approach in a virtualized environment. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2009), IEEE, pp. 519–524.

[31] JOHNSON, T., CONNORS, D., MERTEN, M., AND HWU, W. Run-time cache bypassing. *IEEE Transactions on Computers 48*, 12 (1999), 1338–1354.

[32] KANG, H., AND WONG, J. L. To hardware prefetch or not to prefetch?: A virtualized environment study and core binding approach. *SIGPLAN Not. 48*, 4 (2013), 357–368.

[33] KASERIDIS, D., STUECHELI, J., AND JOHN, L. K. Bank-aware dynamic cache partitioning for multicore architectures. In *International Conference on Parallel Processing (ICPP)* (2009), IEEE, pp. 18–25.

[34] KASTURE, H., AND SANCHEZ, D. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014), ACM, pp. 729–742.

[35] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[36] KOH, Y., KNAUERHASE, R., BRETT, P., BOWMAN, M., WEN, Z., AND PU, C. An analysis of performance interference effects in virtual environments. In *IEEE International Symposium on Performance Analysis of Systems & Software* (2007), IEEE, pp. 200–209.

[37] LARABEL, M., AND TIPPETT, M. Phoronix test suite. http://www.phoronix-test-suite.com/, 2011. Accessed May, 2015.

[38] LAZAR, A. A., AND SEMRET, N. Design and analysis of the progressive second price auction for network bandwidth sharing. *Telecommunication Systems—Special issue on Network Economics* (1999).

[39] LEE, H., CHO, S., AND CHILDERS, B. R. Cloudcache: Expanding and shrinking private caches. In *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)* (2011), IEEE, pp. 219–230.

[40] LIN, W.-F., AND REINHARDT, S. K. Predicting last-touch references under optimal replacement. Tech. rep., University of Michigan, 2002. CSE-TR-447-02.

[41] LO, D., CHENG, L., GOVINDARAJU, R., RANGANATHAN, P., AND KOZYRAKIS, C. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ACM, pp. 450–462.

[42] LU, Q., LIN, J., DING, X., ZHANG, Z., ZHANG, X., AND SADAYAPPAN, P. Soft-OLP: Improving hardware cache performance through software-controlled object-level partitioning. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2009), IEEE, pp. 246–257.

[43] LUCIER, B., LEME, R. P., AND TARDOS, E. On revenue in the generalized second price auction. In *Proceedings of the 21st International Conference on the World Wide Web (WWW)* (2012), ACM, pp. 361–370.

[44] MAILLÉ, P., AND TUFFIN, B. Multi-bid auctions for bandwidth allocation in communication networks. In *IEEE INFOCOM* (2004).

[45] MOVSOWITZ, D., AGMON BEN-YEHUDA, O., AND SCHUSTER, A. Attacks in the resource-as-a-service (RaaS) cloud context. In *Distributed Computing and Internet Technology*, N. Bjørner, S. Prasad, and L. Parida, Eds., vol. 9581 of *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 10–18.

[46] MURALIDHARA, S. P., SUBRAMANIAN, L., MUTLU, O., KANDEMIR, M., AND MOSCIBRODA, T. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)* (2011), ACM, pp. 374–385.

[47] NATHUJI, R., KANSAL, A., AND GHAFFARKHAH, A. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)* (2010), ACM, pp. 237–250.

[48] NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. Fair queuing memory systems. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39)* (2006), IEEE, pp. 208–222.

[49] OU, Z., ZHUANG, H., NURMINEN, J. K., YLÄ-JÄÄSKI, A., AND HUI, P. Exploiting hardware heterogeneity within the same instance type of Amazon EC2. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2012).

[50] PIQUET, T., ROCHECOUSTE, O., AND SEZNEC, A. Exploiting single-usage for effective memory management. In *Asia-Pacific Computer Systems Architecture Conference* (2007), Springer, pp. 90–101.

[51] POZO, R., AND MILLER, B. Scimark 2.0. http://math.nist.gov/scimark2, 2000. Accessed January, 2016.

[52] QURESHI, M. K., JALEEL, A., PATT, Y. N., STEELY, S. C., AND EMER, J. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)* (2007), ACM, pp. 381–391.

[53] RAFIQUE, N., LIM, W. T., AND THOTTETHODI, M. Architectural support for operating system-driven cmp cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)* (2006), ACM, pp. 2–12.

[54] RAJ, H., NATHUJI, R., SINGH, A., AND ENGLAND, P. Resource management for isolation enhanced cloud services. In *Proceedings of the ACM Workshop on Cloud Computing Security (CCSW)* (2009), ACM, pp. 77–84.

[55] SANCHEZ, D., AND KOZYRAKIS, C. Vantage: Scalable and efficient fine-grain cache partitioning. *ACM SIGARCH Computer Architecture News 39*, 3 (2011), 57–68.

[56] SCOLARI, A., SIRONI, F., SCIUTO, D., AND SANTAMBROGIO, M. D. A survey on recent hardware and software-level cache management techniques. In *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2014), IEEE, pp. 242–247.

[57] SHARIFI, A., SRIKANTAIAH, S., MISHRA, A. K., KANDEMIR, M., AND DAS, C. R. METE: Meeting end-to-end QoS in multicores through system-wide resource management. In *Proceedings of the ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2011), ACM, pp. 13–24.

[58] SOARES, L., TAM, D., AND STUMM, M. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)* (2008), IEEE Computer Society, pp. 258–269.

[59] TAYLOR, G., DAVIES, P., AND FARMWALD, M. The TLB slice a low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)* (1990), ACM, pp. 355–363.

[60] VICKREY, W. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance 16*, 1 (1961), 8–37.

[61] WALDSPURGER, C. A., HOGG, T., HUBERMAN, B. A., KEPHART, J. O., AND STORN, W. S. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering 18*, 2 (1992), 103–117.

[62] WANG, X., WEN, X., LI, Y., LUO, Y., LI, X., AND WANG, Z. A dynamic cache partitioning mechanism under virtualization environment. In *IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)* (2012), IEEE, pp. 1907–1911.

[63] YAROM, Y., GE, Q., LIU, F., LEE, R. B., AND HEISER, G. Mapping the Intel last-level cache. Tech. rep., Cryptology ePrint Archive, Report 2015/905, 2015.

[64] YE, C., BROCK, J., DING, C., AND JIN, H. Rochester elastic cache utility (RECU): Unequal cache sharing is good economics. *International Journal of Parallel Programming* (2015), 1–15.

[65] YE, Y., WEST, R., CHENG, Z., AND LI, Y. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT)* (2014), ACM, pp. 381–392.

[66] ZHANG, X., DWARKADAS, S., AND SHEN, K. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)* (2009), ACM, pp. 89–102.

Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads using Simultaneous Multithreading

Xi Yang[†] Stephen M. Blackburn[†] Kathryn S. McKinley[‡]
[†]*Australian National University* [‡]*Microsoft Research*

Abstract

Web services from search to games to stock trading impose strict Service Level Objectives (SLOs) on tail latency. Meeting these objectives is challenging because the computational demand of each request is highly variable and load is bursty. Consequently, many servers run at low utilization (10 to 45%); turn off simultaneous multithreading (SMT); and execute only a single service — wasting hardware, energy, and money. Although co-running batch jobs with latency critical *requests* to utilize multiple SMT hardware contexts (lanes) is appealing, unmitigated sharing of core resources induces non-linear effects on tail latency and SLO violations.

We introduce *principled borrowing* to control SMT hardware execution in which batch threads borrow core resources. A batch thread executes in a reserved batch SMT lane when no latency-critical thread is executing in the partner request lane. We instrument batch threads to quickly detect execution in the request lane, step out of the way, and promptly return the borrowed resources. We introduce the `nanonap` system call to stop the batch thread's execution without yielding its lane to the OS scheduler, ensuring that requests have exclusive use of the core's resources. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when request lane is idle improves utilization between 90% and 25% on one core depending on load, without compromising request SLOs. Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

1 Introduction

Latency-critical web services, such as search, trading, games, and social media, must consistently deliver low-latency responses to attract and satisfy users. This requirement translates into Service Level Objectives (SLOs) governing latency. For example, an SLO may include an average latency constraint and a *tail constraint*, such as that 99% of requests must complete within 100 ms [6, 7, 13, 34]. Many such services, such as Google Search and Twitter [6, 8, 18], systematically underutilize the available hardware to meet SLOs. Furthermore, servers often execute only one service to ensure that latency-critical requests are free from interference. The result is that server utilizations are as low as 10 to 45%. Since these services are widely deployed in large numbers of datacenters, their poor utilization incurs enormous commensurate capital and operating costs. Even small improvements substantially improve profitability.

Meeting SLOs in these highly engineered systems is challenging because: (1) requests often have variable computational demands and (2) load is unpredictable and bursty. Since computation demands of requests may differ by factors of ten or more and load bursts induce queuing delay, overloading a server results in highly non-linear increases in tail-latency. The conservative solution providers often take is to significantly over-provision.

Interference arises in chip multiprocessors (CMPs) and in simultaneous multithreading (SMT) cores when contending for shared resources. A spate of recent research explores how to predict and model interference between different workloads executing on distinct CMP cores [8, 23, 25, 28], but these approaches target and exploit large scale diurnal patterns of utilization, e.g., co-locating batch workloads at night when load is low. Lo et al. explicitly rule out SMT because of the highly unpredictable and non-linear impact on tail latency (which we confirm) and the inadequacy of high-latency OS scheduling [23]. Zhang et al. do not attempt to reduce SMT-induced overheads, but rather they accommodate them using a model of interference for co-running workloads [35]. Their approach requires ahead-of-time profiling of all co-located workloads and over-provisioning. Prior work lacks dynamic mechanisms to monitor and control batch workloads on SMT with low latency.

This research exploits SMT resources to increase utilization without compromising SLOs. We introduce *principled borrowing*, which dynamically identifies idle cycles and borrows these resources. We implement borrowing in the ELFEN[1] scheduler, which co-runs batch threads and latency-critical requests, and meets request SLOs. Our work is complementary to managing shared cache and memory resources. We first show that latency-critical workloads impose many idle periods and they are short. This result confirms that scheduling at OS granu-

[1] In the Grimm fairy tale, *Die Wichtelmänner*, elves borrow a cobbler's tools while he sleeps, making him beautiful shoes.

larities is inadequate, motivating fine-grain mechanisms.

ELFEN introduces mechanisms to control thread execution and a runtime that monitors and schedules threads on distinct hardware contexts (*lanes*) of an SMT core. ELFEN pins latency-critical requests to one SMT lane and batch threads to $N-1$ partner SMT lanes on a N-way SMT core. A *batch* thread monitors a paired lane reserved for executing latency-critical requests. (We use '*requests*' for concision.) The simplest *borrow idle* policy in ELFEN ensures mutual exclusion — requests execute without interference. Batch threads monitor the request lane and when the request lane is occupied, they release their resources. When the request lane is idle, batch threads execute. We introduce `nanonap`, a new system call, that disables preemption and invokes `mwait` to release hardware resources quickly — within ~3 000 cycles. This mechanism provides semantics that neither yielding, busy-waiting, nor `futex` offer. After calling `nanonap`, the batch thread stays in this new kernel state without consuming microarchitecture resources until the next interrupt arrives or the request lane becomes idle. These semantics ensure that requests incur no interference from batch threads and pose no new security risks. Since the batch thread is never out of the control of the OS, the OS may preempt it as needed. The shared system state that ELFEN exploits is already available to applications on the same core, and ELFEN reveals no additional information about co-runners to each other.

We inject scheduling and profiling mechanisms into batch applications at compile-time. A binary re-writer could also implement this functionality. The instrumentation frequently checks for a request running on the paired SMT lane by examining a shared memory location. When the batch thread observes a request, it immediately invokes `nanonap` to release hardware resources. This policy ensures that the core is always busy, but it only utilizes one SMT lane on a two-way SMT core at a time.

More aggressive borrowing policies use both lanes at once by giving batch threads a budget that limits overheads imposed on requests, ensuring that SLOs are met. The budget is shaped by the SLO, the batch workload's impact on the latency-critical workload, and the length of the request queue. These policies monitor the request in various ways, via lightweight fine-grain profiling [32].

We implement ELFEN in the Linux kernel and in compile-time instrumentation that self-schedules batch workloads, using both C and Java applications, demonstrating generality. For our latency-sensitive workload, we use the widely deployed Apache Lucene open-source search engine [3]. Prior work shows Lucene has performance characteristics and request demand distributions similar to the Bing search engine [10]. We evaluate ELFEN co-executing a range of large complex batch workloads on two-way SMT hardware. On one core, ELFEN's borrow idle policy achieves peak utilization with essentially no impact on Lucene's 99th percentile latency SLO. ELFEN improves core utilization by 90% at low load and 25% at high loads compared to a core dedicated only to Lucene requests. It consistently achieves close to 100% core utilization, the peak for this policy — one of the two hardware contexts always busy. On an eight core CMP, the borrow idle policy usually has no impact or slightly improves 99th percentile latency because cores never go to sleep. Occasional high overheads at high load may require additional interference detecting techniques. Improvements in CMP utilization are more substantial than for one core because at low load, many cores may be idle. ELFEN consistently achieves close to 100% of the no-SMT peak, which is also the borrow idle policy's peak utilization.

Choosing a policy depends on provider workloads, capacity, and server economics, including penalties for missed SLOs and costs for provisioning servers. Providers currently provide excess capacity for load bursts and SLOs slack for each request. Our approach handles both. Our most conservative borrow idle policy steps out of the way during load bursts and suits a setting where the penalties for missed SLOs are very high. Our more agressive policies can soak up slack and handle load bursts. They offer as much as two times better utilization but at the cost of higher median latencies and higher probability of SLO violations. For server providers with latency-critical and batch workloads, the main benefit of our work is to substantially reduce the required number of servers for batch workloads.

In summary, contributions of this paper include:

- analysis of why latency-critical workloads systematically underutilize hardware and the opportunities afforded by idle periods;
- `nanonap`, a system call for fine-grain thread control;
- ELFEN, a scheduler that borrows idle cycles from underutilized SMT cores for batch workloads without interfering with latency-critical requests;
- a range of scheduling policies;
- an evaluation that shows ELFEN can substantially increase processor utilization by co-executing batch threads, yet still meet request SLOs; and
- an open-source implementation on github [33].

Our approach requires only a modest change to the kernel and no changes to application source code, making it easy to adopt in diverse systems and workloads.

2 Background and Motivation

We motivate our work with workload characteristics of latency-critical services; the non-linear effects on latency from uncontrolled interference with SMT; the opportunity to improve utilization availed by idle resources; and

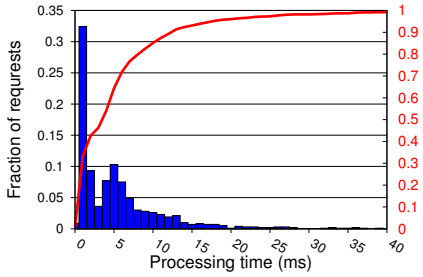


Figure 1. Highly variable demand is typical for latency-critical workloads. Lucene demand distribution with request processing time on x-axis in 1 ms buckets, fraction of total on left y-axis, and cumulative distribution red line on right y-axis.

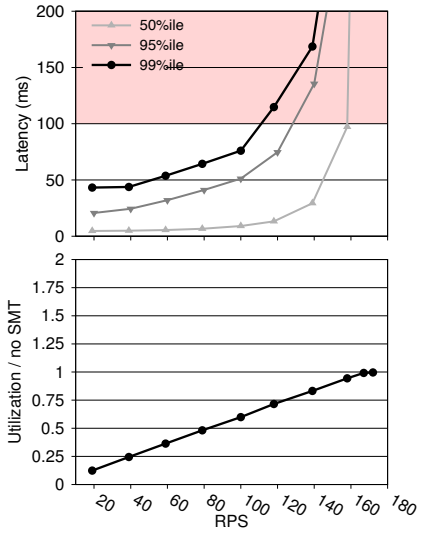


Figure 2. Overloading causes non-linear increases in latency. Lucene percentile latencies and utilization on one core. Highly variable demand induces queuing delay, which results in non-linear increases in latency.

requirements on responsiveness dictated by idle periods. Section 4 describes our evaluation methodologies.

Processing demand The popular industrial-strength Apache Lucene enterprise search engine is our representative service [3]. Prior work shows that services such as Bing search, Google search, financial transactions, and personal assistants have similar computational characteristics [6, 8, 10, 12, 29]. Figure 1 plots the distribution of request processing times for Lucene executing in isolation on a single hardware context. The bars (left y-axis) show that most requests are short, but a few are very long. This high variance of one to two orders of magnitude is common in such systems.

Load sensitivity This experiment shows that high load induces non-linear increases on latency. We assume a 100 ms service level objective (SLO) on 99th percentile latency for requests. A front end on separate hardware issues search requests at random intervals following an exponential distribution around the prescribed requests per second (RPS) mean rate. As soon as Lucene completes a request, it processes the next request in the queue. If no requests are pending, it idles. We show results for a single Lucene worker thread running on one core.

Figure 2 shows Lucene percentile latencies and utilization as a function of RPS only on one lane of a two-way SMT core using one Lucene task. The two graphs share the same x-axis. The top graph shows median, 95th, and 99th percentile latency for requests, the bottom graph shows CPU utilization which is the sum of the fraction of time the lanes are busy normalized to the theoretical peak for a system with SMT disabled. The maximum utilization is 2.0, but the utilization in Figure 2 never exceeds 1.0 because only one thread handles requests, so only one lane is used. As RPS increases, median, 95th, and 99th percentile latencies first climb slowly and then quickly accelerate. The 99th percentile hits a wall when RPS rise above 120 RPS, while the request lane utilization is only 70% at 120 RPS, leaving the two-way SMT core substantially underutilized when operating at a *practical* load for a 100 ms 99th percentile tail latency target.

Random request arrival times and the high variability of processing times combine to produce high variability in queuing times and non-linear increases in latencies at high RPS. As we show next, adding a co-runner on the same core using SMT has the effect of throttling the latency-critical workload, effectively moving to the right in Figure 2. Movements to the right lead to increasingly unpredictable latencies, and likely violations of the SLO.

Simultaneous Multithreading (SMT) This section gives SMT background and shows that simultaneously executing requests on one lane of a 2-way SMT core and a batch thread on the other lane degrades request latencies. This result confirms prior work [8, 23, 35] and explains why service providers often disable SMT. We measure core idle cycles to show that the opportunity for improvement is large, if the system can exploit short idle periods.

We illustrate the design and motivation of SMT in Figure 3. Figure 3(a) shows that when only one thread executes on a core at a time, hardware resources such as the issue queue and functional units are underutilized (white). Figure 3(b) shows two threads sharing an SMT-enabled core. The hardware implements different sharing policies for various resources. For example, instruction issue may be performed round-robin unless one thread is unable to issue, and the load-store queue partitioned in half, statically, while other functional units are shared fully dynamically. It is important to note that such policies

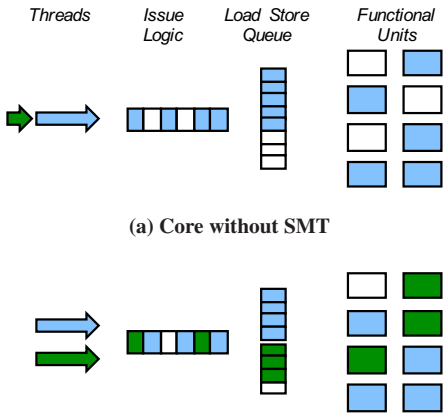


Figure 3. Simultaneous Multithreading (SMT) A single thread often underutilizes core resources. SMT dynamically shares the resources among threads.

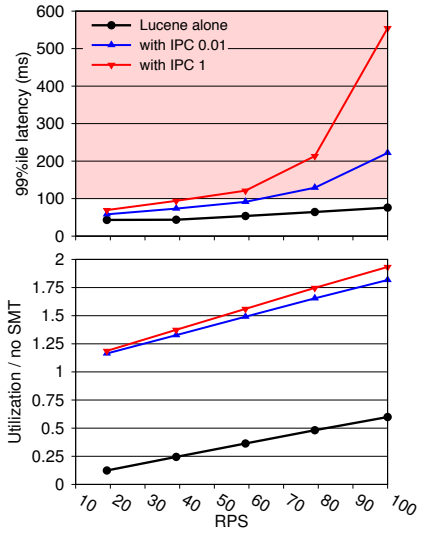


Figure 4. Unfettered SMT sharing substantially degrades tail latency. Lucene 99th percentile latency and lane utilization with IPC 1 and IPC 0.01 batch workloads.

mean that a co-running thread consumes considerable core resources *even when that thread has low IPC*.

To measure lower bounds on SMT interference, we consider two microbenchmarks as batch workloads executing on an Intel 2-way SMT core. The first uses a non-temporal store and memory fence to continuously block on memory, giving an IPC of 0.01. For instance, the Intel PAUSE instruction has a similar IPC. The other performs a tight loop doing nothing (IPC=1) when running alone. Neither consume cache or memory resources. Figure 4 shows the impact of co-running batch workloads on the 99% percentile latency of requests and lane utilization. Utilization improves over no co-runner significantly since the batch thread keeps the batch lane busy, but request latency degrades substantially, even when the batch thread has very low resource requirements (IPC = 0.01). For instance, at 100 RPS without a co-runner, 99th percentile latency is 76 ms. RPS must fall to around 40 RPS to meet the same 99th percentile latency with a low IPC co-runner.

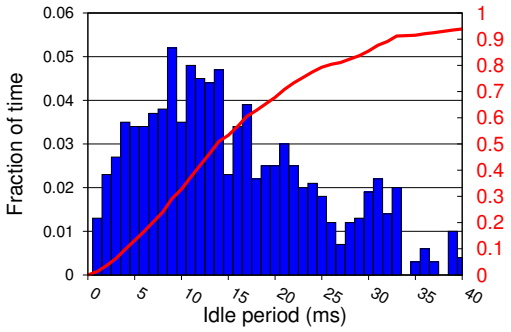


Figure 5. Lucene Inter-request idle times are highly variable and are frequently short. Histogram (left y-axis) shows the distribution of request idle times. The line (right y-axis) shows the cumulative time between completing one request and arrival of the next request at 0.71 lane utilization on one core at RPS = 120.

Co-running moves latencies to the right on RPS curves, into the steep exponential, with devastating effect on SLOs. Because SMT hardware shares resources such as issue logic, the load store queue (LSQ), and store buffers, tail latency suffers even when the batch workload has an IPC as low as 0.01. If a request is short, a co-runner may substantially slow it down without breaching SLOs. Unfortunately request demand is not known a priori. Moreover, request demand is hard to predict [15, 17, 19, 24] and the prediction is never free or perfect, thus we do not consider request prediction further.

To increase utilization without imposing any degradations on latency-critical requests cannot use multiple SMT lanes simultaneously. The strategies we explore are thus (1) to enforce mutual exclusion, executing a batch thread only when the partner lane is idle (*borrow idle*), and (2) to give the batch thread a budget for how much it may overlap execution with requests. These strategies require observing requests, detecting idle periods, and controlling batch threads.

Idle cycle opportunities Now we explore the frequency and length of idle periods to understand the requirements on the granularity of observing requests and controlling batch threads. Figure 5 shows the fraction of all idle time (y-axis) due to periods of a given length (x-axis). The histogram (blue) indicates the fraction of all idle time due to idle times of a specific period, while the line (red) shows a cumulative distribution function. For example, this shows that 2.3% of idle time is contributed by idle

times of 15 ms in length (blue), and 53% of total idle time is due to idle times of 15 ms or less (red). **Highly variable and short idle periods dictate low-latency observation and control mechanisms.**

3 ELFEN Design and Implementation

This section describes the design and implementation of ELFEN, in which latency-critical requests and batch threads execute in distinct SMT hardware contexts (*lanes*) on the same core to improve server utilization. Given an N-way SMT, $N-1$ SMT lanes execute batch threads, *batch lanes*, and one SMT lane executes latency-critical requests, the *request lane*. We restrict our exposition below to 2-way SMT for simplicity and because our evaluation Intel hardware is 2-way SMT.

As Figure 4 shows, unfettered interference on SMT hardware quickly leads to SLO violations. ELFEN controls batch threads to limit their impact on tail latency. We explore borrowing policies applying the principle of either eliminating interference or limiting it based on some budget. The simplest policy enforces mutual exclusion by forcing batch threads to relinquish their lane resources whenever the request lane is executing a request. More aggressive borrowing policies add overlaping the execution of batch threads and requests, governed by a budget.

The ELFEN design uses two key ideas: (1) high-frequency, low-overhead monitoring to identify opportunities, and (2) low-latency scheduling to exploit these opportunities. The implementation instruments batch workloads at compile time with code that performs both monitoring and self-scheduling. The simple borrow-idle policy requires no change to the latency-critical workload. More aggressive policies require the latency-critical framework to expose the request queue length and a current request identifier via shared memory. Batch threads use `nanonap` to release hardware resources rapidly without relinquishing their SMT hardware context.

Our current design assumes an environment consisting of a single latency-critical workload, and any number of instrumented batch workloads. (Scheduling two or more distinct latency-critical services simultaneously on one server is a different and interesting problem that is beyond our scope.) Our instrumentation binds threads to cores with `setaffinity()` to force all request threads onto the identifiable request lane and batch threads onto partner batch lane(s). The underlying OS is free to schedule batch threads on batch lanes. Each batch thread will then fall into a monitoring and self-scheduling rhythm.

3.1 Nanonap

This section introduces the system call `nanonap` to monitor and schedule threads at a fine granularity. The key semantics `nanonap` delivers is to put the hardware context to sleep *without* releasing the hardware to the OS scheduler. We first explain why existing mechanisms, such as `mwait`, `WRLOS`, and `hotplug` do not directly deliver the necessary semantics.

The `mwait` instruction releases the resources of a hardware context with low latency. This instruction is available in user-space on SPARC and is privileged on x86. The IBM PowerEN user-level `WRLOS` instruction has similar semantics [26]. Calls to `mwait` are normally paired with a `monitor` instruction that specifies a memory location that `mwait` monitors. The OS or another thread wakes up the sleeping thread by writing to the monitored location or sending it an interrupt. The Linux scheduler uses `mwait` to save energy. It assigns each core a privileged idle task when there are no ready tasks. Idle tasks call `mwait` to release resources, putting the hardware in to a low-power state. Unfortunately, simply building upon any of these mechanisms in *user space* is insufficient because the OS may, and is likely to, schedule other ready threads to the released hardware context. In contrast, because it disables preemption, `nanonap` ensures that no other thread runs on the lane, releasing all hardware resources to its partner lane.

Another mechanism that seems appealing, but does not work, is `hotplug`, which prohibits any task from executing in specified SMT lanes. The OS first disables interrupts, moves all threads in the lane(s), including the thread that invoked `hotplug`, to other cores, and switches the lane(s) to the idle task which then calls the `mwait` instruction. While `hotplug` moves threads off a lane to other cores, user-space calls such as `futex` yield the lane, so other threads may execute in it. Therefore, neither the `hotplug` interface nor user-space locking nor calls to `mwait` are designed to release and acquire SMT lanes to and from each other because a thread does not *retain exclusive ownership of a lane while it pauses*.

We design a new system call, `nanonap`, to control the SMT microarchitecture hardware resources directly. Any application that wants to release a lane invokes `nanonap`, which enters the kernel, disables preemption, and sleeps on a per-CPU `nanonap` flag. From the kernel's perspective, `nanonap` is a normal system call and it accounts for the thread as if the thread is still executing. Because `nanonap` does not disable interrupts and the kernel does not preempt the thread that invoked the `nanonap`, the SMT lane stays in a low-power sleep state until the OS wakes the thread up with an interrupt or the ELFEN scheduler sets the `nanonap` flag. After the SMT lane wakes up, it enables preemption and returns from the system call. Figure 6 shows the pseudocode of `nanonap`, which we implement as a wrapper that invokes a virtual device using the Linux OS's `ioctl` interface for devices.

No starvation or new security state The `nanonap` system call and monitoring of request lanes do not cause starvation or pose additional opportunities for security breaches. Starvation does not occur because `nanonap`

does not disable interrupts. The scheduler may wake up any napping threads and schedule a different batch thread on the lane at the end of a time quanta, as usual. When a batch thread wakes up or a new one starts executing, it tests whether its request lane partner is occupied and if so, puts itself to sleep. Since the OS accrues CPU time to batch threads waiting due to a `nanonap`, user applications cannot perform a denial of service attack simply by continuously calling `nanonap`, since the OS will schedule a napping thread away after they exhaust their time slice.

The ELFEN instrumentation monitors system state to make decisions. It reads memory locations and performance counters that reveal if the core has multiple threads executing. All of this system state is *already* available to threads on the same core — ELFEN reveals no additional information about co-runners to each other.

3.2 Latency of Thread Control

This section presents an experiment that measures the latency of sleeping and waking up with `nanonap`, `mwait`, and `futex`. Measuring these latencies is challenging because detecting exactly when a lane releases hardware resources must be inferred, rather than measured directly.

When a batch thread executes `mwait` on our Intel hardware, the lane first enters the shallow sleeping C1 state immediately. If no other thread executes in the lane for a while, it then enters a deep sleep state and releases its hardware resources to the active request lane. We measure how long it takes the lane to enter the deep sleeping state indirectly as follows. The CPU executes a few μops to transition an SMT lane from the shallow to the deep sleep state. For measurement purposes, we thus configure the measurement thread to continuously record how many μops the measurement thread has retired and how many μops the whole core retires every 150 cycles. When the measurement thread notices that the sleeping SMT lane does not retire any μops for a while, then retires a few more μops, and then stops retiring μops, it infers that the SMT lane is in the deep sleep state.

Figure 7 shows a microbenchmark that measures the latencies of sleeping and waking up with `nanonap`, `mwait`, and `futex`. The microbenchmark has two threads: a measurement thread and another thread, T2. The measurement thread puts T2 to sleep and wakes it up. The two threads execute on the same core but different SMT lanes. The measurement thread sets a flag, forcing T2 to sleep (line 5). T2 then executes `sleep` which either calls `nanonap` or `futex` to put the SMT lane to sleep, according to which is being measured. The `wait_until_t2_goes_to_sleep()` (line 6) call performs the deep sleep detection process described in the above paragraph. We measure wake-up latency directly (lines 10 to 13). The measurement thread sets a flag (line 11) and then detects when T2 starts executing instructions (line 12).

```
1  /****************** USER ********************/
2  void nanonap() {
3    ioctl(/dev/nanonap);
4  }
5  /****************** KERNEL ******************/
6  nanonap virtual device: /dev/nanonap;
7  per_cpu_variable: nap_flag;
8  ioctl(/dev/nanonap) {
9    disable_preemption();
10   my_nap_flag = this_cpu_flag(nap_flag);
11   monitor(my_nap_flag);
12   mwait();
13   enable_preemption();
14 }
```

Figure 6. Pseudo code for `nanonap`.

```
1  /***** MEASUREMENT THREAD ON ONE SMT LANE *****/
2  void measure() {
3    /* measure send-to-sleep latency */
4    start_sleep_request = timestamp();
5    ask_t2_sleep();
6    wait_until_t2_goes_to_sleep();
7    finish_sleep_request = timestamp();
8
9    /* measure wake-up latency */
10   start_wakeup_request = timestamp();
11   wakeup_t2();
12   wait_until_t2_wakes_up();
13   finish_wakeup_request = timestamp();
14
15   if (measuring_futex || measuring_nanonap) {
16     sleep_latency =
17       finish_sleep_request - start_sleep_request;
18     wakeup_latency =
19       finish_wakeup_request - start_wakeup_request;
20   }
21   if (measuring_mwait) {
22     sleep_latency = finish_sleep_request - mwait_start;
23     wakeup_latency = mwait_finish - start_wakeup_request;
24   }
25 }
26 /************ T2 ON OTHER SMT LANE ************/
27 void sleep() {
28   if (measuring_futex)
29     wait_on_futex();
30   else if (measuring_nanonap || measuring_mwait)
31     nanonap();
32 }
33 void nanonap() {
34   ...
35   mwait_start = timestamp();
36   monitor(flag);
37   mwait();
38   mwait_finish = timestamp();
39   ...
40 }
```

Figure 7. Microbenchmark that measures time to sleep with `nanonap`, `mwait`, and `futex`.

We execute each configuration 100 times. Figure 8 presents the time and the 95% confidence interval for using `nanonap`, `mwait`, and `futex` to sleep and wake-up a thread executing in a partner SMT lane. The time to put a lane to sleep for `mwait` is 2 443 cycles, is 3 285 cycles for `nanonap`, and is 11 518 cycles for `futex`, 3.5 times slower than `nanonap`. Waking up a lane directly with `mwait` takes 1 036 cycles — essentially the hardware latency of wake-up. The latency of `nanonap`'s wake-up is

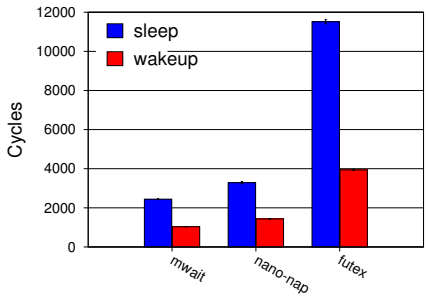


Figure 8. Time to sleep and wake-up SMT partner lanes.

similar to `mwait`'s at 1438 cycles. However, `futex` takes 3 968 cycles, which is 2.72 times slower than `nanonap`. Although `futex` is substantially slower, this latency is likely tolerable, since most idle periods are more than 1 million cycles on our 2 GHz machine (1 ms in Figure 5). However, as explained above, neither the semantics of locks nor user-space calls to `mwait` are adequate for our purposes.

3.3 Continuous Monitoring and Signaling

Sleeping and waking up fast is necessary but not sufficient. The scheduler has to know when to act. We further exploit the `nanonap` mechanism to improve over our SHIM [32] fine-grain profiling tool. SHIM views time-varying software and hardware events as continuous 'signals' (in the signal processing sense of the word). Rather than using interrupts to examine request threads, as many profiling tools do, we configure our batch threads to continuously reads signals from memory locations and hardware performance counters to *profile* request threads. In the simplest case, the profiling observes whether the request thread is executing. Our prior work shows that SHIM accurately observes events at granularities as fine as 100s to 1000s of cycles with overheads of a few percent when executing on another core. However, when threads share an SMT core, we saw similar overheads from SMT sharing as shown in Figure 4. In this paper, we use the `nanonap` mechanism to essentially eliminate this overhead.

Whereas SHIM observes signals from a *dedicated thread*, here we (1) use GCC `-pg` instrumentation [9] to insert checks at method prologues into *C batch workloads* and (2) piggyback on the default Java VM checks at every method entry/exit and loop backedge [22]. These mechanisms add minimal overhead as shown by Lin et al. [22] and, most importantly, remove the need for a third profiling thread to observe request threads.

At each check, the fast-path consists of a few instructions to check monitored signals. For efficiency, this fast path is inlined to the body of compiled methods. If the observed signal matches the condition (e.g., the scheduler sets the memory location that indicates the request lane is idle), the batch thread jumps to an out-of-line function to handle the task of putting itself to sleep.

3.4 ELFEN Scheduling

We design and implement four policies that borrow underutilized resources without compromising SLOs.

Borrowing Idle Cycles The simplest way to improve utilization is to run the batch workload *only* when the latency-critical workload is idle. Section 2 analyzed the maximum CPU utilization of Lucene while meeting a practical SLO at ~70% of one SMT lane, which corresponds with prior analysis of latency-critical workloads [6, 8, 10, 12, 29]. Therefore even when the latency-sensitive workload is executing at the maximum utilization at which it can meet SLOs, there is an opportunity to improve utilization by 30% if the batch workload can borrow this excess capacity. At lower loads, there is even more opportunity.

This policy enforces mutual exclusion. Batch threads execute only when the request lane is empty. When a request starts executing, the batch thread immediately sleeps, relinquishing its hardware resources to the latency-critical request. When the request lane becomes idle, the batch thread wakes up and executes in the batch lane.

Figure 9(a) shows the simple modifications to the kernel and batch workloads required to implement this policy. We add an array called `cpu_task_map` that maps a lane identifier to the current running task. At every context switch, the OS updates the map, as shown in `task_switch_to()`. By observing this signal, the scheduler knows which threads are executing in the SMT lanes. At each check, the scheduler determines whether the `idle_task` is executing in the request lane. If the request lane is idle, the scheduler either continues executing the batch thread in its lane or starts a batch thread. If the request lane is occupied, the scheduler immediately forces the batch thread to sleep with `nanonap`.

Fixed Budget Borrowing idle cycles is simple and as we show, effective, but we can further exploit underutilized resources when requests may incur some overhead and still meet their SLO. In particular, short requests, which typically dominate, easily meet the SLO under moderate loads. We consider the maximum slowdown requests can incur under a certain load as a budget for the batch workload. As an example, consider an SLO latency of 100 ms for 99% of requests. If 99% of requests executing exclusively on the core complete in 53 ms at some RPS, then there exists headroom of $100 - 53 = 47$ ms. We thus could take a budget of 47 ms for executing batch tasks. (We leave more sophisticated policies that also incorporate load along the lines of Haque et al. [10] to future work.)

Given a budget, the fixed-budget scheduler will execute

```
/****************** KERNEL ******************/
/* maps lane IDs to the running task */
exposed SHIM signal: cpu_task_map

task_switch(task T) { cpu_task_map[thiscpu] = T; }
idle_task() { // wake up any waiting batch thread
  update_nap_flag_of_partner_lane();
  ......
  mwait();
}
/*************** BATCH TASKS ****************/
/* fast path check injected into method body */
check:
if (!request_lane_idle) slow_path();

slow_path() { nanonap(); }
```

(a) Borrow idle policy.

```
/********** LATENCY CRITICAL WORKLOAD *********/
exposed SHIM signal: queue_len

/*************** BATCH THREADS **************/
per_cpu_variable: lane_status = NORMAL;
per_cpu_variable: start_stamp;
check:
if (request_lane_idle && queue_len == 0) {
  lane_status = NEW_PERIOD;
} else if (!request_lane_idle) {
  slow_path();
}
slow_path() {
  switch (lane_status) {
  case NORMAL:
    nanonap();
    break;
  case NEW_PERIOD:
    lane_status = CO_RUNNING;
    start_stamp = rdtsc();
    break;
  case CO_RUNNING:
    now = rdtsc();
    if (now - start_stamp >= budget)  // expired
      lane_status = NORMAL;
} }
```

(b) Fixed budget policy.

```
/********** LATENCY CRITICAL WORKLOAD *********/
exposed SHIM signals: queue_len, running_request

/*************** BATCH THREADS **************/
/* Same as the fixed budget policy, except... */
per_cpu_variable: last_request
...
 case NEW_PERIOD:
   ...
   last_request = running_request;
   ...
 case CO_RUNNING:
   if (running_request != last_request &&
       queue_len == 0) {
     last_request = running_request;
     start_stamp = rdtsc();
   }
...
```

(c) Refresh budget policy.

```
/*************** BATCH THREADS **************/
/* Same as the refresh budget policy, except... */
...
 case CO_RUNNING:
   ...
   /* calculate IPC of LC lane */
   ratio = ref_IPC / (ref_IPC - LC_IPC)
   real_budget = budget * ratio;
   if (now - start_stamp >= real_budget)
     lane_status = NORMAL;
...
```

(d) Dynamic refresh policy.

Figure 9. The pseudocode of four scheduling policies.

batch threads concurrently with requests in their respective SMT lanes when the scheduler is confident that the batch threads will not slow down any request longer than the given budget. Co-running with a request for T ms slows down the processing time of the request less than T ms. For requests that never wait in the queue, the processing time is the same as the request latency. So, it is safe to co-run with these requests for a budget period. Figure 9(b) shows the implementation of this policy. Line 7 detects when the request queue is empty and renews the budget period, such that the next request will co-execute with the batch thread for the fixed budget.

As we showed in Section 2, the request lane is frequently idle for short periods because after one request finishes there are no pending requests, and most requests are short. The fixed-budget scheduler only uses its budget when a new request that never waits in the queue starts executing in the request lane. When the scheduler detects that a new request starts executing and the `lane_status` is set to `NEW_PERIOD` because the request queue was empty before this requests started, it co-schedules the batch thread in its lane for the budget period. If the request is finished in the period and there are no waiting requests, the scheduler resets the budget and uses it for the next request. When the budget expires, the scheduler puts the batch thread to sleep. When another idle period begins because the request terminates, the request queue is empty, and no other request is executing, the scheduler restarts the batch thread and repeats this process. Note that if N requests execute in quick succession without idle gaps, this simple scheduler only co-executes the batch thread with the first request that begins after an idle period. This conservative strategy ensures that each request is only impacted for the budget period of its execution.

Refresh Budget The refresh budget policy extends the fixed budget policy based on the observation that once a request has completed *and* the queue is empty, the budget may be refreshed. The rationale is that the original budget was calculated based on avoiding a slowdown that could prevent the just-completed task from meeting the SLO. Once that task completes, then the budget may be recalculated with respect to the *new* task meeting the SLO. However, because the slowdown imposed by the batch workload is imparted not just on the running request, but on all requests behind it in the queue, we only refresh the budget if the task changes *and* the queue is empty. Figure 9(c) shows the code.

Dynamic Budget The dynamic budget policy is the most aggressive policy and builds upon the refresh budget policy. It uses a *dynamic budget* that is continuously adjusted according to the base budget and the IPC of the latency-critical request. This policy requires us first to profile the IPC with no interference and then to monitor the impact of co-running on request IPC. We implement

the monitoring based on the sampling ideas in SHIM [32]. We read the IPC hardware performance counter of the request lane from the batch lane, at high frequency with low overhead. When the latency-critical request's IPC is high, it will be proportionately less affected by the batch workload, so we adjust the dynamic budget accordingly.

4 Methodology

Hardware & OS We use a 2.0 GHz Intel Xeon-D 1540 Broadwell [16] processor with eight two-way SMT cores, a 12 MB shared L3. Each core has a private 256 KB L2, a 32 KB L1D and a 32 KB L1I. The TurboBoost maximum frequency is 2.6 Ghz, TDP is 45 W. The machine has 16 GB of memory and two Gigabit Ethernet ports. We disable deep sleep and TurboBoost.

We use Ubuntu 15.04, Linux version 4.3.0, and the perf subsystem to access the hardware performance counters. We implement the `nanonap` mechanism as a virtual device as shown in Figure 6. We modify the idle task to wake up sleeping batch lanes as shown in Figure 9(a). We expose a memory buffer to user space to determine which tasks are running on which cores.

Latency-Critical Workload We use the industrial-strength open-source Lucene framework to model behavior similar to the commercial Bing web search engine [10] and other documented latency-critical services [6, 8, 12, 29]. Load variation results from both the number of documents that match a request and from ranking calculations. We considered using `memcached`, a key-value store application, because it is an important latency-critical workload for Facebook [11, 27] and a popular choice in the OS and architecture communities. However, each request offers the same uniformly very low demand ($<$10 K instructions) [11], which means requests saturate the network before they saturate the CPU resources on many servers. Recent work offers OS and hardware solutions to these network scalability problems [4, 21], which we believe if combined with our work would be complementary. We leave such investigations to future work.

We execute Lucene (svn r1718233) in the Open JDK 64 bit server VM (build 25.45-b02, mixed mode). We use the Lucene performance regression framework to build indexes of the first 10 M files from Wikipedia's English XML export [31] and use 1141 term requests from `wikimedium.10M.nostopwords.tasks` as the search load. The indexes are small enough to be cached in memory on our machine. We warm up the server before running any experiments.

We send Lucene requests from another machine that has the same specifications as the server. The two machines are connected via an isolated Gigabit switch. For each experiment, we perform 20 invocations. For each invocation, the client loads 1141 requests, shuffles the requests, and sends requests 5 times. The client issues search requests at random intervals following an exponential distribution around the prescribed RPS mean rate. We report the median result of the 20 invocations. The 95% confidence interval is consistently lower than $\pm$0.02.

Batch Workloads We use 10 real-world Java benchmarks from the DaCapo 2006 release [5] and three micro C benchmarks, `Loop`, `Matrix`, and `Flush`. The DaCapo benchmarks are popular open-source applications with non-trivial memory loads that have active developers and users. Using DaCapo as batch workloads represents a real world setting. The C micro benchmarks demonstrate the generality of our approach and give us control over the interference pattern. `Loop` calls an empty function and has an IPC of 1. It consumes front-end pipeline resources. `Matrix` calls a function that multiplies a 5$\times$5 matrix, a computationally intensive high IPC workload. It consumes both front-end pipeline and functional-unit resources. `Flush` calls a function that zeros a 32 KB buffer, a disruptive co-runner that flushes the L1D cache.

We run Java benchmarks with JikesRVM [1], release 3.1.4 + git commit fd68163, a Java-in-Java high performance Virtual Machine, using a large 200 MB heap. The JIT compiler in JikesRVM already inserts checkpoints for thread control and garbage collection into function prologues, epilogues and loop back-edges. We add to these a check for co-runner state, as shown in Figure 9. For C micro benchmarks, we use GCC's `-pg` instrumentation option [9] to add checks to method prologues.

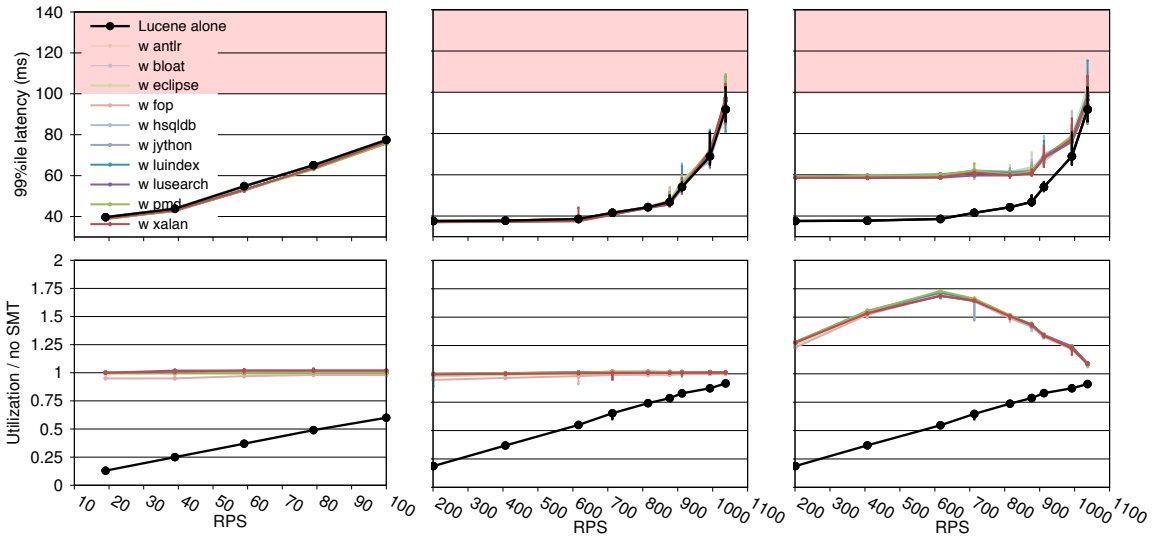
Measurements We use a target, 100 ms request latency for 99% of requests, as our SLO in all of our experiments, which is a practical SLO target for the search engine.

5 Evaluation

This section evaluates the ability of ELFEN to improve server utilization while meeting Service Level Objectives (SLOs) and ELFEN overheads.

Borrow idle We first present ELFEN configurations that use the *borrow idle* policy with DaCapo as the batch workload. This policy minimizes the impact on request latencies. Figure 10(a) plots latency (top) and utilization (bottom) versus requests per second (RPS) on the x-axis for Lucene without (black) and with each of the ten DaCapo batch workloads (colors) executing on one two-way SMT core of the eight-core Broadwell CPU. Figure 10(b) presents these same configurations executing seven instances of each DaCapo benchmark on seven cores. The eighth core manages network communication (receiving requests and returning results), queuing, and starting requests for the latency-sensitive workload. We plot median latency; error bars indicate 10th and 90th percentiles.

The results in Figure 10(a) and 10(b) show that *executing these batch workloads in idle cycles imposes very little impact on Lucene's SLO on a single core or a CMP.*



(a) Borrow idle, one core **(b)** Borrow idle, seven cores **(c)** Dynamic refresh budget, seven cores

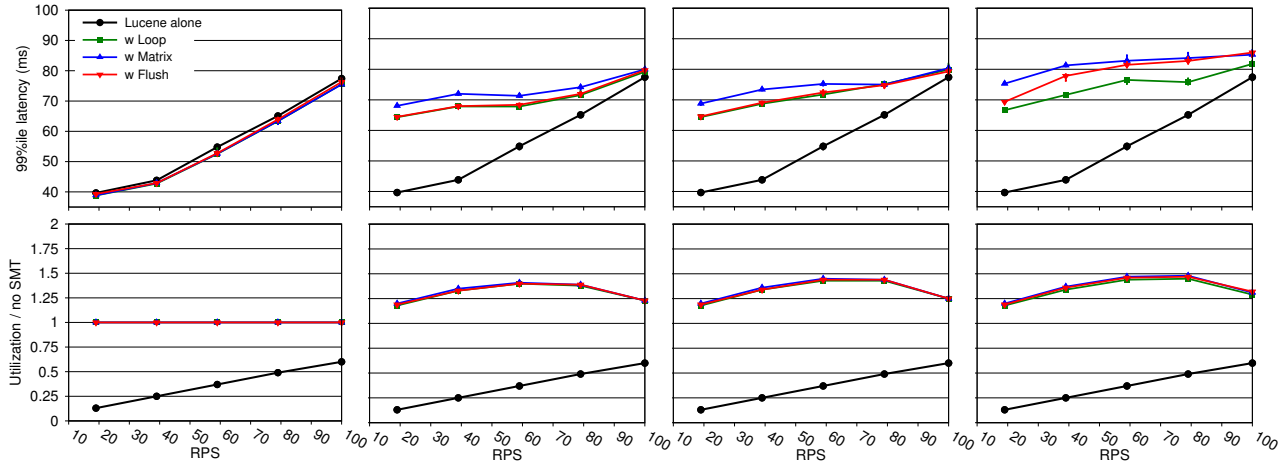
Figure 10. 99th percentile latency (top) and utilization (bottom) for Lucene co-running with DaCapo batch workloads.

ELFEN achieves essentially the same 99th percentile latency at the same requests per second (RPS) with or without batch execution. In fact on one core, ELFEN sometimes delivers slightly lower latencies for Lucene when executing each of the batch workloads in the other lane during idle periods. This results occur because running the batch thread in the other lane causes the core never to enter any of its sleep states. When a new request arrives, the core is guaranteed not to be sleeping, its request lane is empty, and thus the core will service requests slightly faster. With the borrow idle policy, the peak utilization of the core is 100% out of 200% since each core has 2 hardware contexts, but by design, only one is active at a time. Because ELFEN keeps the core busy, executing requests as they arrive in one lane and batch threads with mutual exclusion in the other, it often achieves its peak potential of 100% utilization, but when the utilization of the batch workload is low, the total utilization may be less than 100%.

The chip multiprocessor (CMP) results in Figure 10(b) show better throughput scaling than just a factor of seven. For example, at 60 ms, the single core system can sustain about 70 RPS, while the seven-core system can sustain as much as 1000 RPS. Remember that most requests are short, and long requests contribute most to tail latencies. CMPs better tolerate long request latencies than a single core by executing multiple other short requests on other cores, so fewer short requests incur queuing delay when a long request monopolizes a core. At moderate loads, we again see some improvements to request latency when co-running with batch workloads because the cores never sleep, whereas cores are sometimes idle long enough without co-runners to sleep. However, continuously and fully utilizing all seven cores on the chip incurs more interference, and thus we see some notable degradations in the 99%ile latency at high load. There are two sources of increased latency. First, the effects of managing the queue and request assignment, which shows some non-scalable results. For example, even small amounts of contention for the request queue impacts tail latency independent of ELFEN. ELFEN slightly exacerbates this problem. Second, as prior work has noted and addressed [14, 23, 25], requests and batch threads can contend for shared chip-level resources on CMPs, such as memory and bandwidth. Adding such techniques to ELFEN should further improve its effectiveness.

Increasing Utilization on a Budget Figure 11 presents latency (top graphs) and utilization (bottom) for the four ELFEN scheduling policies described in Section 3: borrow idle, fixed budget, refresh budget, and dynamic refresh on one core. The budget-based policies all borrow idle cycles and trade latency for utilization, slowing the latency-critical requests to increase utilization. Comparing the top row in the figure shows that increasingly aggressive policies cause more degradations in the 99th percentile latency. In these RPS ranges, Lucene's requests meet the 100 ms SLO latency target, but are degraded.

Borrowing idle cycles and co-executing batch threads with requests increases utilization significantly. Comparing across the utilization figures reveals that the budget-based policies further improve utilization compared to borrowing idle cycles. Core utilization rises as load in-



(a) Borrow idle **(b)** Fixed budget **(c)** Refresh budget **(d)** Dynamic refresh

Figure 11. 99th percentile latency (top) and utilization (bottom) for Lucene co-running with C microbenchmarks under four ELFEN policies on a single two-way SMT core.

creases. At moderate loads, ELFEN achieves utilizations over 1.4 for the *fixed budget* policy and 1.5 for the *dynamic refresh* policy. All budget-based policies achieve utilizations of at least 1.2. When the system becomes highly loaded with requests, ELFEN adjusts by executing the co-runners less, and thus total utilization drops. While all of the ELFEN policies are effective at trading off utilization for SLOs, the most aggressive *dynamic refresh* policy consistently runs at higher utilization. The *dynamic refresh* policy is performing precise, fine grain monitoring of request IPC to more accurately and effectively manage this tradeoff. Although we study IPC, ELFEN may monitor and partition other resources, such as memory and cache. Although higher utilization is appealing, some service providers may not be willing to sacrifice throughput of latency-critical tasks, so for them the most practical policy may be to borrow idle cycles.

Figure 10(c) shows the latency and utilization results for the most aggressive dynamic refresh policy on our CMP with DaCapo as the batch workload. This policy degrades the 99th percentile latency by 20 ms before reaching a peak utilization of 1.75 at around 600 RPS. At larger RPS, ELFEN schedules the batch less, system utilization drops and the latency approaches to the same level of the borrow idle policy.

Overhead on Batch Workload Overhead on the batch workload comes from instrumentation, interference with the latency-sensitivity requests, and being frequently paused and restarted. As we pointed out above, Lin et al. [22] show the instrumentation overheads are low, at most a couple percent.

Figure 12 measures these other overheads. It presents the execution time, user time utilization, and user level IPC of each DaCapo benchmark co-running with Lucene normalized to its execution alone on one core. When co-running, we use the borrow idle policy and load the Lucene at 80 RPS, which leads to about 50% utilization for both Lucene and each DaCapo benchmark. The execution time of co-running each DaCapo benchmark increases by 49% on average as predicted by the 50% utilization. There are small variations in these slowdowns, but none of them are due to DaCapo programs executing more instructions when co-running — the number of retired instruction at user level is the same. Furthermore, DaCapo does not execute instructions less efficiency, because IPC decreases are only 1%.

Variation in execution times is due to variations in utilization already present in the DaCapo benchmarks. If the batch workload is idle for some other reason (e.g., waiting on I/O or a lock), then a request that forces it to stop executing will affect it less. The more idle periods the batch workload has, the less execution is degraded. This effect causes normalized execution time and utilization to be strongly correlated. For instance, the *pmd* benchmark incurs the largest slowdown in execution time, 59%, and the largest utilization reduction, 36%. The *fop* benchmark has the lowest native utilization in these benchmarks. Consequently, it has both the smallest slowdown and the smallest utilization reduction, 47% and 26%.

6 Related Work

Exploiting SMT Lo et al. [23] demonstrate that naively co-running batch workloads with latency-critical workloads violates Google's SLO, even under light load. They show that for many latency-critical workloads, uncon-

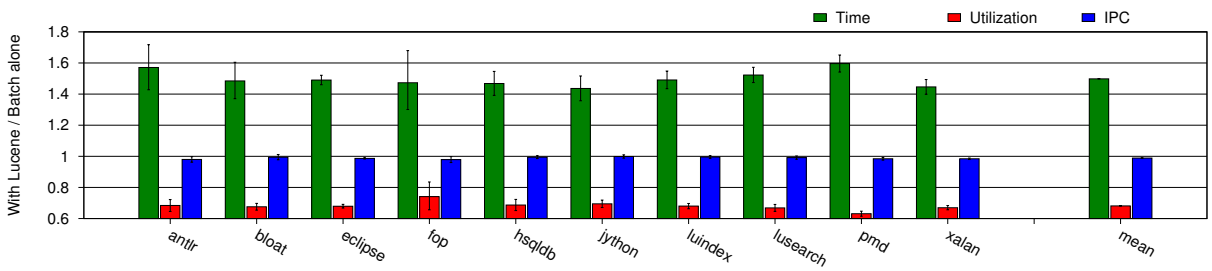


Figure 12. Normalized DaCapo benchmark execution time, user space CPU utilization, and IPC.

trolled interference due to SMT co-location is unacceptably high, and conclude that it is not practical to share cores with SMT. Our results contradict this conclusion.

Ren et al. [29] exploit SMT for a server that exclusively handles latency-sensitive requests (no co-location) and thus requires over-provisioning to handle load spikes. Herdrich et al. [14] note that achieving latency objectives with current SMT hardware is challenging because the shared resources introduce contention that make it hard to reason about and meet SLOs. They propose SMT rate control as a building block to improve fairness and determinism in SMT, which dynamically partitions resources and implements biased scheduling. These mechanisms should help limit interference on requests and complement our approach. They do not evaluate latency-critical workloads, seek to borrow idle cycles, or offer a fine-grain thread-switching mechanism, as we do here.

Accommodating Overheads Zhang et al. [35] use offline profiling of batch workloads to precisely predict the overhead due to co-running with latency-critical requests on SMT. They then carefully provision resources to co-run batch workloads whilst maintaining SLOs for latency-critical workloads. Unlike our work, they do not attempt to minimize the overhead of co-running batch workloads. Rather, they predict and then accommodate it. They measured interference due to co-run batch workloads in the range of 30%-50%.

POSIX Real-Time Scheduling Leverich and Kozyrakis [20] propose using POSIX real-time scheduling disciplines to prioritize requests over co-run batch threads. When hardware contexts are scarce, this approach ensures that latency-critical requests have priority — batch threads will be the first to block. When given sufficient hardware contexts however, the approach does not control for interference due to co-running. Thus it does not address the problem we address here: avoiding interference due to co-running while utilizing SMT.

Exposing and Evaluating `mwait` Anastopoulos and Koziris [2] use `mwait` to release resources to another SMT thread when waiting on a lock. Wamhoff et al. [30] make `mwait` user-level visible and then use it to put cores into sleep states so as to provide power headroom for DVFS to boost performance on other cores which are executing threads on the program's critical path. They measure the latency of putting an entire core into a C1 sleep state on an Intel Haswell 4 770 and found that it was 4 655 cycles. This result is broadly consistent with our measurements, which are for a single hardware context on a more recent processor. With regard to semantics, Meneghin et al. [26] claim fine-grain thread communication requires user-level mechanisms, whereas we offer an intermediate point that involves the OS, but not the OS scheduler. None of this prior work has the same semantics as `nanonap` for hardware control, which we exploit for both fine-grain monitoring and scheduling.

7 Conclusion

This paper shows how to use SMT to execute latency-critical and batch workloads on the same server to increase utilization *without* degrading the SLOs of the latency-critical workloads. We show, given a budget, how to control latency degradations to further increase utilization while meeting SLOs. Our policies borrow idle cycles and control interference by reserving one lane for requests and one for batch threads. By reserving SMT lanes, ELFEN always immediately executes the next request when the previous one completes or a new one arrives. Using low-overhead monitoring and `nanonap`, ELFEN responds promptly to release core resources to requests or to control interference from batch threads. Our principled borrowing approach is extremely effective at increasing core utilization. Whereas current systems achieve utilizations of 5% to 35% of a 2-way core (by only using one lane at 10% to 70%) while meeting SLOs, ELFEN's *borrow idle* policy uses both lanes to improve utilization at low load by 90% and at high load by 25%, delivering consistent and full utilization of a core at the same SLO. On CMPs, ELFEN with the borrow idle policy is extremely effective as well, achieving its peak utilization without degrading SLOs for all but the highest loads. No prior work has managed this level of consistent server utilization without degrading SLOs.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2):399–418, 2005.

[2] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, April 2008. doi: 10.1109/IPDPS.2008.4536358.

[3] Apache Lucene. `http://lucene.apache.org/`, 2014.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 49–65, 2014.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[6] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[8] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, 2014.

[9] GCC. Program instrumentation options, 2016. URL `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html#Instrumentation-Options`.

[10] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–175, 2015.

[11] S. Hart, E. Frachtenberg, and M. Berezecki. Predicting memcached throughput using simulation and modeling. In *Symposium on Theory of Modeling and Simulation (IMS/DEVS)*, pages 40:1–8, 2012.

[12] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–238, 2015.

[13] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, page 12, 2012.

[14] A. Herdrich, R. Illikkal, R. Iyer, R. Singhal, M. Merten, and M. Dixon. SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms. In *HotPar*, pages 219–230, 2013.

[15] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[16] Intel. Intel Xeon processo d-1540, 12m cache, 2.00 GHz, 2013. URL `http://ark.intel.com/products/87039/Intel-Xeon-Processor-D-1540-12M-Cache-2_00-GHz`.

[17] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *SIGCOMM '13*, 2013.

[18] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, pages 158–169, 2015.

[19] S. Kim, Y. He, S.-W. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2015.

[20] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality of service. In *ACM European Conference on Computer Systems (Eurosys)*, 2014.

[21] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[22] Y. Lin, K. Wang, S. M. Blackburn, M. Norrish, and A. L. Hosking. Stop and Go: Understanding yieldpoint behavior. In *ACM International Symposium on Memory Management (ISMM)*, pages 70–80, 2015.

[23] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, pages 450–462, 2015.

[24] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 50–61, 2001.

[25] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[26] M. Meneghin, D. Pasetto, H. Franke, F. Petrini, and J. Xenidis. Performance evaluation of interthread communicationmechanisms on multicore/multithreaded architectures, 2012. URL `http://researcher.watson.ibm.com/researcher/files/ie-pasetto_davide/PerfLocksQueues.pdf`. 2 page version in ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2012.

[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[28] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 219–230, 2013.

[29] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity in interactive services. In *ACM International Conference on Autonomic Computing (ICAC)*, pages 45–58, 2013.

[30] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The TURBO diaries: Application-controlled frequency scaling explained. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 193–204, 2014.

[31] Wikipedia. Wikipedia:database download, 2016. URL `https://en.wikipedia.org/wiki/Wikipedia:Database_download`. Accessed January 2016.

[32] X. Yang, S. M. Blackburn, and K. S. McKinley. Computer performance microscopy with Shim. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, pages 170–184, 2015.

[33] X. Yang, S. M. Blackburn, and K. S. McKinley. ELFEN scheduler open source implementation, June 2016. URL `https://github.com/elfenscheduler`.

[34] J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, pages 257–266, 2008.

[35] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS prediction on real-system smt processors to improve utilization in warehouse scale computers. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 406–418, 2014.

Coherence Stalls or Latency Tolerance: Informed CPU Scheduling for Socket and Core Sharing

Sharanyan Srikanthan Sandhya Dwarkadas Kai Shen
Department of Computer Science, University of Rochester
{srikanth,sandhya,kshen}@cs.rochester.edu

Abstract

The efficiency of modern multiprogrammed multicore machines is heavily impacted by traffic due to data sharing and contention due to competition for shared resources. In this paper, we demonstrate the importance of identifying latency tolerance coupled with instruction-level parallelism on the benefits of colocating threads on the same socket or physical core for parallel efficiency. By adding hardware counted CPU stall cycles due to cache misses to the measured statistics, we show that it is possible to infer latency tolerance at low cost. We develop and evaluate SAM-MPH, a multicore CPU scheduler that combines information on sources of traffic with tolerance for latency and need for computational resources. We also show the benefits of using a history of past intervals to introduce hysteresis when making mapping decisions, thereby avoiding oscillatory mappings and transient migrations that would impact performance. Experiments with a broad range of multiprogrammed parallel, graph processing, and data management workloads on 40-CPU and 80-CPU machines show that SAM-MPH obtains ideal performance for standalone applications and improves performance by up to 61% over the default Linux scheduler for mixed workloads.

1 Introduction

Modern multi-socket multicore machines present a complex challenge in terms of performance portability and isolation, especially for parallel applications in multiprogrammed workloads. Performance is heavily impacted by traffic due to data sharing and contention due to competition for shared resources, including physical cores, caches, memory, and the interconnect.

Significant effort has been devoted to mitigating the impact on performance of competition for shared resources [7, 15, 17, 18, 23, 26] for applications that do not share data. Our own past work, Sharing-Aware Mapper (SAM) [22], has shown that inter-socket coherence activity among threads can be used as a criterion for same-socket thread colocation for improved system efficiency and parallel application performance. Using existing x86 performance counters, SAM is able to separate inter-socket traffic due to coherence from that due to memory access, favoring colocation for threads experiencing high coherence traffic, and distribution for threads with high cache and/or memory bandwidth demand. Although SAM is able to infer inter-socket coherence traffic, it cannot determine the impact of the coherence activity on application performance. This inability to relate traffic to performance limits its effectiveness to prioritize tasks in multiprogrammed workloads.

In this paper, we develop and evaluate a multicore CPU scheduler that combines information on sources of traffic with tolerance for latency and need for computational resources. First, we demonstrate the importance of identifying latency tolerance in determining the benefits of colocating threads on the same socket or physical core on parallel efficiency in multiprogrammed workloads. High inter-socket coherence activity does not translate to proportional benefit from thread colocation for different applications or threads within an application. The higher the latency hiding ability of the thread, the lower the impact of inter-socket coherence activity on performance. We infer the benefits of colocation using commonly available hardware performance counters, in particular, CPU stall cycles on cache misses. A low value for CPU stall cycles is an indication of latency tolerance, making stall cycles an appropriate metric for prioritizing threads for colocation.

Second, we focus on the computational needs of individual threads. Hyperthreading [13], where hardware computational contexts share a physical core, present complex tradeoffs for applications that share data. Colocating threads on the same physical core allows direct access to a shared cache thereby resulting in low latency data communication when fine-grain sharing (sharing of cache-resident data) is exhibited across threads. At the same time, competition for functional units can reduce the instructions per cycle (IPC) for the individual threads relative to running on independent physical cores. The benefits of colocation are therefore a function of granularity of sharing (whether the reads and writes by different threads occur while the data is still cache resident) as well as the instruction-level parallelism available within

each thread. We find that a combination of IPC and coherence activity thresholds are sufficient to inform this tradeoff.

Finally, we show that utilizing interval history in phase classification can avoid the oscillatory and transient task migrations that may result from merely *reacting* to immediate past behavior [8]. In particular, we keep track of the level of coherence activity that was incurred by a thread in prior intervals, as well as its tolerance for latency, and use this information to introduce hysteresis when identifying a phase transition.

The combination of these three optimizations enable us to obtain ideal performance for standalone applications and improve performance by up to 61% over linux for multiprogrammed workloads. Performance of multiprogrammed workloads improve on average by 27% and 43% over standard Linux on 40- and 80-CPU machines respectively. When compared with SAM [22], our approach yields an average improvement of 9% and 21% on the two machines with a peak improvement of 24% and 27%. We also reduce performance disparity across the applications in each workload. These performance benefits are achieved with very little increase in monitoring overhead.

2 Background: Separating Traffic due to Sharing and Memory Access

This work builds on our prior effort of SAM [22], a Sharing-Aware-Mapper that monitors individual task [1] behavior using hardware performance counters. SAM identifies and combines information from commonly available hardware performance counter events to separate traffic due to data sharing from that due to memory access. Further, the non-uniformity in traffic is captured by separately characterizing intra- and inter-socket coherence activity, and local versus remote memory access.

Following an iterative, interval-based approach, SAM uses the information about individual task traffic patterns to retain colocation for tasks with high intra-socket coherence activity, and consolidate tasks with high inter-socket coherence activity. At the same time, SAM distributes tasks with high memory bandwidth needs, colocating them with the memory they access. These decisions reduce communication and contention for resources by localizing communication whenever possible.

While SAM is able to separate and identify coherence traffic from memory bandwidth needs, it does not currently determine the impact of the traffic on performance; in other words, its ability to tolerate the latency of communication, which would allow task prioritization for constrained resources. Additionally, SAM currently does not differentiate between logical hardware contexts and physical cores. Furthermore, while SAM's successive iterations are able to capture changes in application behavior and workload mixes to effect task placement changes, it merely *reacts* to the current state of task placement to effect a more efficient task placement. Thus, it misses opportunities to learn from past placement decisions as well as to adapt to periodicity in application behavior. Our goal in this paper is to address these shortcomings in SAM and realize multiprogrammed performance much closer to standalone static best placement.

[1] In this paper, a task refers to an operating system-level schedulable entity such as a process or a thread.

3 Identifying Latency Tolerance and Computational Needs

In this section, we demonstrate the importance of identifying latency tolerance and computational needs in multicore task placement, and show how this information may be inferred from widely available performance counter events.

3.1 Tolerance for Coherence Activity Latency

Data sharing across tasks can result in varying communication latencies primarily dictated by task placement. The closer the tasks sharing the data, the lower the latency. For example, when tasks that share data are placed across sockets, the need to move data across sockets results in substantially increased latency. Hence, the natural choice would be to prioritize tasks with high coherence activity for colocation on the same socket.

However, the performance impact of coherence activities depends in reality on the latency tolerance of the application. We focus here on identifying this latency tolerance in addition to being able to measure and identify data sharing.

We introduce two additional metrics to augment inter-socket coherence activity as a measure of sharing behavior: IPC (Instructions per Cycle) and SPC (Stalls per inter-socket Coherence event). The Intel platforms provide access to a counter that tracks cycles stalled on all last-level cache misses. While these stalls include those due to coherence activity, they also include stalls on other forms of misses. When coherence activity is high, stalls are dominated by coherence activity, and thus stalls on cache misses can be used as an approximation of stalls due to coherence misses. SPC is thus approximated to be stalls on all last-level cache misses divided by the number of coherence events in the specific time interval.

For low to moderate coherence activity levels, the above approximation for SPC can no longer be justified. In such cases, we use IPC as an indicator of latency tolerance. Higher IPC is generally achieved with high

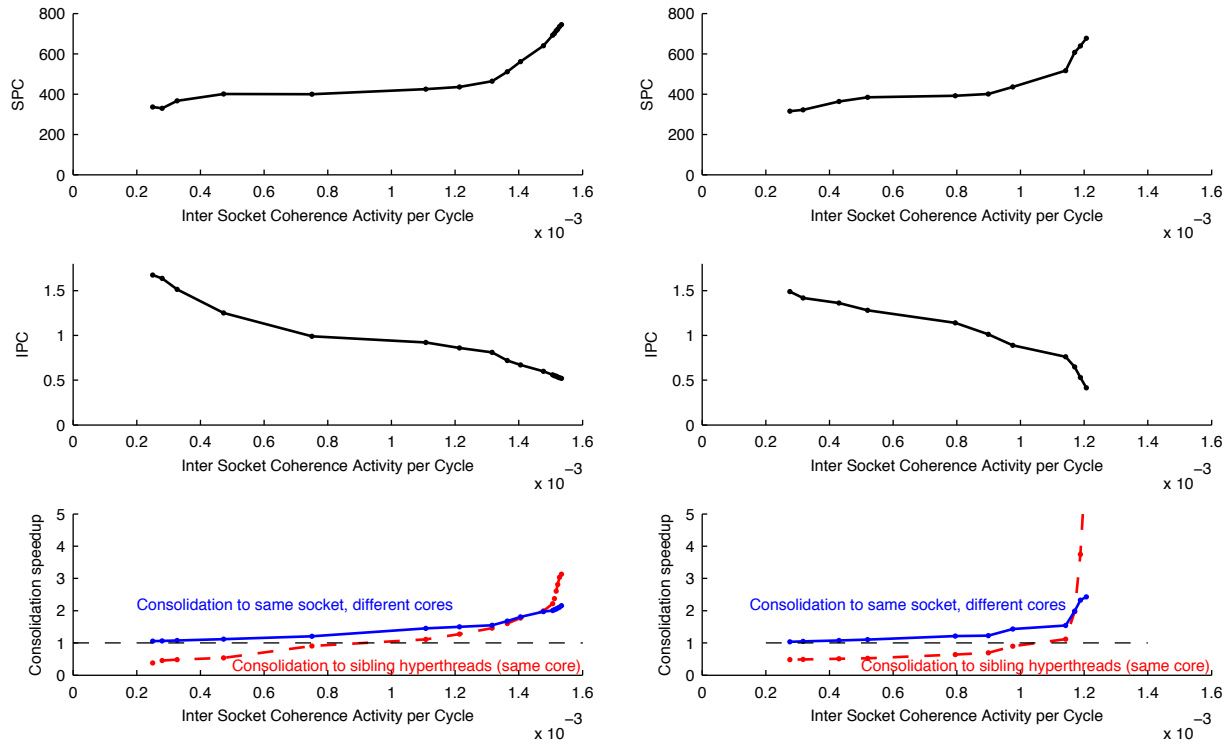


Figure 1: SPC (stalls per coherence event), IPC (instructions per cycle), and speedup (relative to running on separate sockets) when consolidating threads onto the same socket on different physical cores (blue curve) and same physical core (red dashed curve), as a function of inter-socket coherence activity controlled by a microbenchmark with a (left) higher instruction-level parallelism (ILP) and (right) lower ILP computational loop.

instruction-level parallelism, which provides the ability to hide access latency.

In general, higher SPC implies lower latency hiding potential; higher IPC implies more instruction-level parallelism that can be used to hide latency. We use a combination of IPC and SPC in order to be able to predict a task's latency hiding potential. In order to demonstrate the importance of SPC and IPC in making placement decisions, we designed a microbenchmark that is parameterized to control coherence activity. Coherence activity is generated by a *coherence activity* loop that alternates increments by two threads to a set of shared counters that are guaranteed to fit in the level of cache closest to the processor. The frequency of coherence activity is controlled by adding to the *coherence activity* loop a computational loop consisting of (a) additions on registers or (b) additions/multiplications on independent registers thereby increasing instruction-level parallelism, and varying the number of iterations of this loop. Figure 1 presents SPC, IPC, and speedup obtained by consolidating threads on the same socket (or physical core) relative to running on different sockets as the ratio of computation to coherence activity in each outer loop is varied using the two different computational loops. As coherence activity (inter-socket coherence events per cycle) increases, speedup from consolidation mirrors SPC, but the performance gains and inflection points depend on the application. For example, at roughly the same coherence activity of 1.2×10^{-3} coherence events/cycle, consolidation on different physical cores in the same socket results in a speedup of 1.5 for the microbenchmark on the left and 2.4 for the one on the right. The corresponding SPC is 436 and 677, respectively, allowing clear prioritization of the microbenchmark on the right for consolidation.

3.2 Placement on Hyperthreads

Modern processors are often equipped with hyperthreads or other equivalent logical hardware contexts that share the processor pipeline and private caches. Hyperthreads improve processor occupancy and efficiency by providing multiple instruction streams/hardware contexts to keep functional units busy. At the same time, since the hardware contexts share pipeline resources and private caches, contention for these resources can slow the performance of the individual contexts, presenting performance isolation challenges.

Placing tasks that share data on hyperthreads that share

a physical core allows shared data to be accessed directly from the private caches (without traffic on the intra- and inter-socket interconnects), with the potential for a significant performance boost. This advantage is conditional on the data being retained in the cache until the time of access by the sharing task, requiring temporal proximity of the accesses to the shared data.

Parameter Thresholds: Figure 1 shows the impact on performance for different task placement strategies. When coherence activity is moderate ($< 0.78 \times 10^{-3}$) and IPC is sufficiently high (> 0.9), colocating tasks on different physical cores in the same socket results in the best performance. The resource contention introduced by hyperthreading results in slowdowns that are not overcome by the potential for direct access to shared data (the shared counter) from the private cache shared by the tasks. Hence, if placement on the same socket requires sharing physical cores, distributing tasks across sockets works better than colocating them on hyperthreads. However, when coherence activity increases further and IPC is less than 0.9, the benefits of colocating tasks that share data exceed the cost of contention. At very high levels of data sharing, indicated by high coherence activity ($> 0.78 \times 10^{-3}$) and SPC values in excess of 550, the benefits of hyperthreading greatly exceed that of using different physical cores on the same socket. In such cases, consolidation on hyperthreads in the same physical core can provide both performance and energy savings using Intel's powerstepping (the latter has not been explored in this paper).

To summarize, when coherence activity is high ($> 0.78 \times 10^{-3}$), (higher) SPC is used to prioritize applications for consolidation and (lower) IPC is used to determine whether hyperthreading is beneficial. When coherence activity is moderate or low ($< 0.78 \times 10^{-3}$), (higher) IPC is used to prioritize distribution over consolidation.

3.3 Setting up Performance Counters

We use two machines to evaluate our performance: a dual-socket IvyBridge and a quad-socket Haswell. Each processor from the two microarchitectures contain 10 physical cores with 2 hardware contexts/hyperthreads each. Each physical core has a private L1 and L2 cache and share a last level L3 cache with other cores in the processor. We use SAM's infrastructure [22] to access the hardware performance counters provided by Intel's PMU (Performance Monitoring Unit). Each hardware context has only four programmable counters in addition to the fixed counters: instructions, cycles, and unhalted cycles. We use time multiplexing across two time periods to sample eight performance counters.

We use the same counters as SAM to monitor intra-socket coherence activity, inter-socket coherence activity, remote DRAM accesses, and overall bandwidth consumption. Additionally, we also monitor the CYCLE_ACTIVITY: STALL_CYCLES_L2_PENDING event to count the stalls on cache accesses. The counts are normalized using unhalted cycles for the interval of their collection and accumulated in a data structure maintained in the task control block.

4 Latency Tolerance- and Sharing-Aware Mapping

4.1 Implementation Mechanisms

Our mapper is implemented as a kernel module that is executed by a daemon process with root privilege. Hardware performance counters are read at every system timer interrupt (tick) and the information gathered is stored in the task control block of the currently running task. Our mapper is invoked every 100ms, at which time data from the currently executing tasks is consolidated into per-application and per-socket data structures. The daemon maintains a per application data structure for currently active (executing) applications in order to allow grouping of tasks belonging to the same application (based on address space). This data structure keeps track of application history, including current application classification. The per-socket data structure used for determining memory bandwidth requirements and remote memory accesses is identical to that used in SAM [22].

Once the mapper decides on task to core mapping, task migration is effected by updating processor affinity. We use the `sched_setaffinity` kernel call to update the processor affinity of a task. Note that task migration takes place at the next Linux scheduling operation and is required only if tasks are not already colocated. Our decision making logic thereby co-exists with other load balancing operations in Linux.

4.2 Data Consolidation

The hardware performance counter values collected and stored in task control blocks are used to infer inter- and intra-socket coherence activity, memory bandwidth utilization, remote memory accesses, and latency tolerance. The inferred values are then used to categorize tasks as having: 1) low, medium, or high coherence activity; 2) low or high memory bandwidth demand; and 3) low or high IPC, based on thresholds as discussed in Section 3.

The per-task information is aggregated to obtain per-socket information on memory bandwidth and inter-socket coherence activity. The per-task information is also used to categorize the parent application as incur-

ring 1) low, medium, or high coherence activity; and 2) low or high IPC, based on the task with the most coherence activity.

4.3 Mode-based Phase Identification

SAM's original adaptation strategy is *reactive* in that changes in application behavior (phase identification) in one interval trigger a potential re-mapping in the next interval. Reactive adaptation works well when application behavior is relatively stable with few transitions. However, frequent phase transitions can potentially lead to oscillating placement decisions with a resulting reduction rather than improvement in performance.

In this paper, we explore the use of history over multiple past intervals [8]. For each application, a history of interval classification — whether the interval was classified as incurring medium or high coherence activity and high compute intensity (IPC) levels — is maintained for the last n intervals. This history is maintained in three 64 bit integers using shift operations (to shift in a 1 when a particular interval exhibits the behavior). Bit masking and counting are used to determine the occurrence count of each of the bottlenecks. If the occurrence count for a bottleneck exceeds a threshold, we identify the application as suffering from the bottleneck in the next interval. The recent inclusion of the `popcnt` instruction in the Intel ISA results in very fast bit counting operations, allowing low overhead examination. The occurrence threshold builds hysteresis into this feedback control loop, thereby preventing oscillatory behavior. In our implementation, we set n to 10 and the occurrence threshold to 6. We analyze the sensitivity of this threshold and its impact on performance in Section 5.4.

If the number of intervals with high coherence activity exceeds the predefined threshold, the current interval is classified as incurring high coherence activity even if the performance counters for the current interval reflect low coherence activity. This strategy for classification helps avoid task migrations due to transient application behavior as well as avoids oscillatory mappings due to frequent phase transitions.

A cumulative count of the stalls due to inter-socket coherence activity, instructions executed, and cycles elapsed since the last phase transition is maintained in order to calculate SPC and IPC. In an interval with low inter-socket coherence activity and high intra-socket coherence activity, accumulation is suppressed in order to retain SPC and IPC information from the interval where the phase transition was detected. The goal is to retain SPC and IPC information gathered during an inter-socket placement (prior to colocation) for the purposes of prioritization. Based on thresholds, if the classification changes, the cumulative counters are reset to the values for the current interval.

4.4 Hyperthread and Latency Tolerance-Aware Mapping Policy

Presuming that all hardware contexts are busy, the mapping task consists of placing m tasks on m hardware contexts so that there is a $1:1$ correspondence between tasks and hardware contexts. Applications in the highest coherence activity phase are prioritized and mapped first. These applications are sorted by their SPC values and scheduled in order. Applications whose SPC values are not known are placed at the end of the list, but are still scheduled ahead of applications with low data sharing. For each application, tasks that share data with each other are selected for colocation by updating their core affinities. If tasks do need colocation, we look for a socket that has not been assigned any task during the current round of mapping. If a sufficient number of cores in a single socket cannot be found, we colocate the threads on the least number of sockets possible.

We then look at applications with moderate levels of activity. They are sorted in order of IPC and applications with the smallest IPC are prioritized for colocation. When we encounter threads with IPC values greater than the IPC threshold (0.9), we alter the mapping to prohibit the threads from sharing the same physical core. If such a situation is unavoidable inside the same socket, we look to other sockets to determine if performance loss can be avoided. Alternately, if the SPC value of high data sharing applications is more than 550, tasks of that application are preferentially placed on hyperthreads to derive benefits from very high data sharing.

5 Evaluation

Our evaluation was conducted on two machines. The first is our development platform and is a dual-socket machine, equipped with 2.2 GHz Intel Xeon E5-2660 v2 processors from the Ivy Bridge architecture. Each socket can accommodate up to 20 hardware contexts on 10 physical cores, sharing a last-level cache of 25MB. Each socket is directly connected to 8GB local DRAM memory, resulting in non-uniform access to a total of 16GB DRAM memory. This machine is labeled *40-CPU IvyBridge*.

The second machine contains four sockets, equipped with 1.9 GHz Intel Xeon E7-4820 v3 processors from the Haswell architecture. Each processor accommodates up to 20 hardware contexts and has up to 64 GB of local DRAM per socket for a total of 256GB DRAM. This machine is labeled *80-CPU Haswell*.

The operating system we use is Fedora 19 and the kernel was compiled using GCC 4.8.2. Linux kernel (ver-

sion 3.14.8) was modified to accommodate the changes needed for our techniques.

We compare the performance of our sharing-aware mapper, SAM-MPH, with that of SAM [22] and default Linux. In order to attribute the performance improvements in SAM-MPH, we also show incremental performance gained due to identifying and prioritizing task placement based on latency tolerance (SAM-M), hyperthreading aware mapping (SAM-H), and using history across multiple intervals to indentify phase changes (SAM-P).

5.1 Benchmarks

We use a combination of microbenchmarks, SPEC-CPU [1], PARSEC [3], and several parallel and graph-based benchmarks in order to evaluate SAM-MPH.

Similar to SAM, we use microbenchmarks to stress the coherence protocol and memory bandwidth. HuBench and LuBench contain pairs of threads sharing data with each other to generate coherence traffic. HuBench generate high coherence activity and is very sensitive to data sharing latencies while LuBench has enough thread-private computation to hide its data sharing latency. MemBench is a memory intensive microbenchmark that uses multiple threads to access thread-private memory in a streaming fashion. These threads saturate memory bandwidth on one socket, therefore benefiting from distribution across sockets to maximize resource utilization.

GraphLab [16] and *GraphChi* [14] are recent application development tools specially suited for graph-based parallel applications. Unlike the SPECCPU and most PARSEC applications, graph-based processing involves considerable data sharing across several workers. These application are also much more dynamic, with tasks actively being created and deleted, and going through phases of computation that are vastly different in characteristics, depending on the type of problem and the active parallelism available in the input data. The unpredictable and dynamic nature of these applications make them good candidates for evaluating the effectiveness of our mapper.

We use a variety of machine learning, data mining, and data filtering applications for our evaluation—TunkRank (Twitter influence ranking), Alternating Least Squares (ALS) [28], Stochastic gradient descent (SGD) [12], Singular Value Decomposition (SVD) [11], Restricted Bolzman Machines (RBM) [10], Probabilistic Matrix Factorization (PMF) [21], Biased SGD [11], and Lossy SDG [11].

In addition to the above workloads, we also evaluate our implementation on MongoDB, a very widely used data management server. The load for MongoDB is generated by YCSB (Yahoo! Cloud Serving Benchmark).

5.2 Standalone Application Evaluation

Figure 2 shows that for most cases, standalone application performance on SAM-MPH is as good and sometimes better than a static optimum schedule. SAM-MPH and the other variants significantly outperform Linux in almost all cases. Linux generally distributes load across sockets and cores while SAM can detect and respond to data sharing and resource contention but not at the hyperthread level. For these standalone applications, SAM is already able to identify and isolate data sharing to achieve close to the best static schedule. SAM-M and SAM-MP thus add little extra benefit. SAM-MPH is able to identify all the bottlenecks exposed by SAM and outperforms it in 5 cases, demonstrating the importance of considering resource contention at the hyperthread level and of eliminating migrations due to transient application behavior.

With LuBench, PMF, RBM, SVD, and ALS, SAM-MPH performs better than SAM. SAM underperforms Linux in the case of LuBench. LuBench with 20 threads incurs non-trivial data sharing, which prompts SAM to colocate the threads if possible. However, when LuBench executes on two hardware contexts on the same physical core, single thread performance is significantly affected due to contention for pipeline resources. Since Linux by default spreads load out across sockets, it avoids the resource contention on hardware contexts. SAM-MPH identifies both data sharing and pipeline resource contention in LuBench and prioritizes pipeline resource contention as the bigger bottleneck in this case.

For applications PMF, SVD, RBM, and ALS, both Linux and SAM perform very close to the static optimum schedule with SAM being slightly faster. However, SAM-MPH outperforms the best static schedule by a significant margin for PMF and a slight margin for the rest. The best static schedule, as the name suggests, does not adapt to dynamic phases in the application. These four applications exhibit phases that share data and phases that contend for pipeline resources when colocated on hyperthreads. Since SAM-MPH is able to identify these different phases of computation and adapt accordingly to the bigger bottleneck, it is able to perform better than the best static schedule.

Figure 3 shows the intra- and inter-socket coherence activity for the standalone applications on the 40-CPU IvyBridge. In general, SAM is able to suppress inter-socket coherence activity slightly better than SAM-MPH. This slight reduction is attributed to SAM's decision making based on a single interval at a time. SAM-MPH relies on past history (consisting of several intervals) to detect application characteristics, resulting in higher hysteresis. The hysteresis has negligible impact on performance. For the five applications discussed

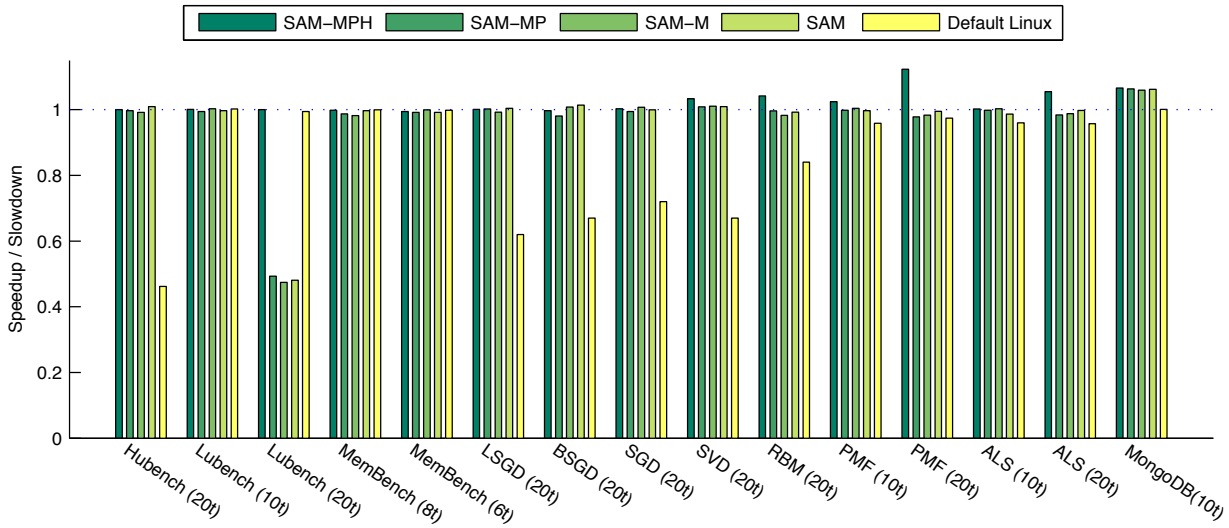


Figure 2: Performance of standalone applications using SAM-MPH, SAM-MP, SAM-M, SAM, and default Linux on the 40-CPU IvyBridge. The performance metric is the execution time speedup (the higher the better) compared to that of the best static task→CPU mapping determined through offline testing.

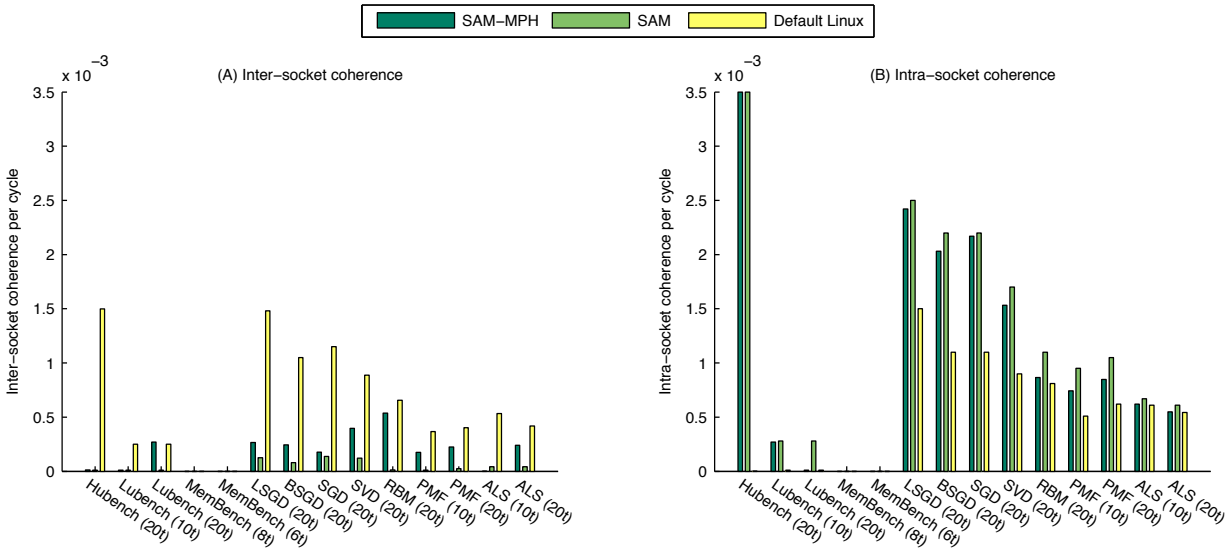


Figure 3: Intra- and Inter-socket traffic for standalone applications. Fig. (A): per-thread inter-socket coherence activity; Fig. (B): per-thread intra-socket coherence activity; All values are normalized to unhalted CPU cycles.

above, SAM-MPH has less intra-socket coherence activity than SAM, since it sometimes avoids colocating threads onto hyperthreads to reduce resource contention on the hyperthreads.

Table 1 outlines information about each application, with major and minor factors that influence its performance. It also shows the Stalls incurred per inter-socket coherence event (SPC). The SPC value reported here is averaged across all high communication phases of the application. We can see for SGD, LSGD, SVD, and BSGD, higher SPC translates to higher performance improvement on colocation.

RBM also exhibits high SPC values but its performance improvement doesn't directly correlate to SPC. This is attributed to the fact that RBM also has compute heavy phases which do not get significant speedup on colocation. Additionally, it would have to be placed on separate physical cores rather than on hyperthreads. Due to these factors, the overall speedup gained during the high coherence phase does not fully translate to very high performance gain.

PMF and ALS have moderate levels of coherence ac-

Application	SPC	Major Bottleneck	Minor Bottleneck
HuBench (20t)	745	DS	None
LuBench (10t)	367	IPC	DS
LuBench (20t)	367	IPC	DS
MemBench (10t)	-	Memory	None
MemBench (20t)	-	Memory	None
SGD (20t)	398	DS	None
BSGD (20t)	421	DS	None
LSGD (20t)	455	DS	None
RBM (20t)	403	DS	IPC
SVD (20t)	442	DS	IPC
PMF (10t)	-	None	IPC and DS
PMF (20t)	-	None	IPC and DS
ALS (10t)	-	None	IPC and DS
ALS (20t)	-	None	IPC and DS

Table 1: Application characteristics and SPC values. DS: Data Sharing: high coherence activity; IPC: Instructions Per Cycle: instruction-level parallelism with high CPU demand; Memory: Memory bound: high memory bandwidth demand.

tivity during which IPC is used for colocation decisions rather than SPC, since SPC cannot be obtained reliably at these levels as explained in Section 3. Hence the SPC value is not reported. When IPC is > 0.9, which is frequently the case for these applications, threads are preferentially placed on separate physical cores in the same socket, with placement across sockets preferred over placement on hyperthreads.

In addition to parallel data sharing workloads, we evaluate SAM-MPH on MongoDB, generating load with YCSB threads, both running on the same machine. For this workload, SAM-MPH and SAM perform very similarly. We observe an improvement of about 3.67% and 6.6% on the larger and smaller evaluation platforms respectively. The marginal improvement is a result of a small amount of data being shared by the threads of the application.

Our experiments with the PARSEC and SPECCPU benchmarks show very similar results to SAM and thus we do not discuss them further in this paper.

Figure 4 shows the results of SAM-MPH, SAM, and the default Linux scheduler on the 80-CPU Haswell. Overall, these results are very similar to results on the 40-CPU IvyBridge: SAM-MPH is able to match and sometimes exceed the performance of the best static schedule. The 80-CPU Haswell, with twice the number of sockets and cores as the 40-CPU IvyBridge, shows the performance gap between SAM-MPH and Linux widening further. SAM-MPH halves the execution time of applications compared to the default Linux scheduler. On average, for standalone workloads, SAM-MPH is 57% faster than Linux. It also matches or exceeds the performance of the best static schedule.

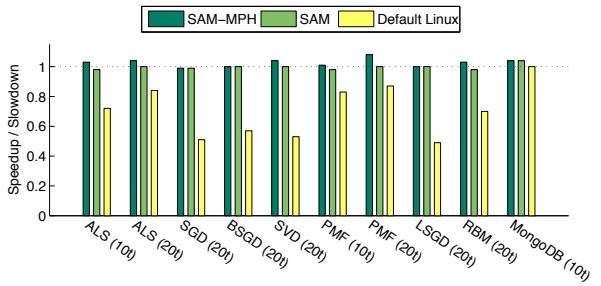


Figure 4: Performance of standalone applications using SAM-MPH, SAM, and default Linux on the 80-CPU Haswell. The performance metric is the execution time speedup (the higher the better) compared to that of the best static task→CPU mapping determined through offline testing.

5.3 Multiprogrammed Workload Evaluation

Applications in multiprogrammed workloads interfere with each other in different ways, depending on the characteristics of applications in the mix and the phase of their execution. This interference may result in slowdown of some or all of the applications.

Table 2 shows various application mixes that are used in the evaluation of SAM-MPH. The workload mixes cover a wide range of application characteristics. Individual applications can be affected due to contention for the processor pipeline, cache space contention, contention for memory bandwidth, communication due to data sharing, and non-uniform communication latencies. We expect SAM-MPH to be able to identify each of these bottlenecks and perform task to core mapping in such a way that would minimize the negative impact on performance due to resource contention and non-uniformity in communication.

Figure 5 shows the performance of SAM-MPH for the multiprogrammed workloads on the 40-CPU IvyBridge. Our performance metric for application mixes is the geometric mean of the individual application speedups, calculated for each application in a workload mix by comparing its runtime to that of its best standalone static runtime.

On average, SAM-MPH is about 27% faster than stock Linux and 9% faster than SAM. More importantly, applications managed by SAM-MPH show very little degradation in performance when compared with the best standalone static schedule. It must be noted that for many workload mixes, it is not possible to get numbers matching the standalone static schedule due to resource contention. The average speedup for SAM-MPH is 0.976, proving that applications seldom show signs of slowdown. The minimum speedup with SAM-MPH is 0.93.

SAM-MPH is able to improve SAM's performance

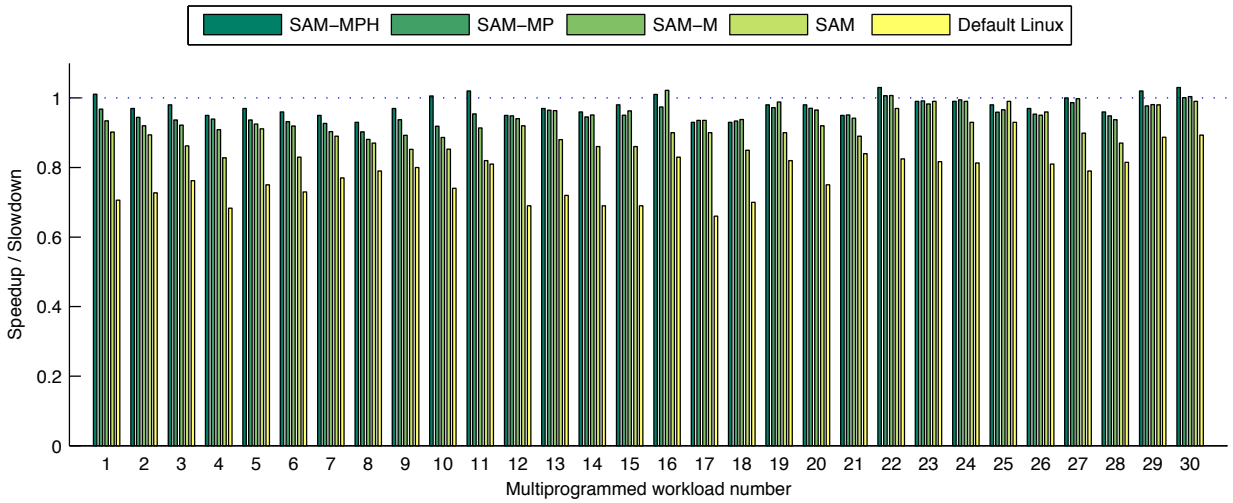


Figure 5: Performance of multiprogrammed workloads using SAM-MPH, SAM-MP, SAM-M, SAM, and default Linux on the 40-CPU IvyBridge. The performance metric is the geometric mean of the individual application speedup (higher is better) compared to execution time obtained for a standalone run using the best static task→CPU mapping determined through offline testing.

primarily due to 2 factors. First, SAM-MPH can prioritize applications that are more sensitive to bottlenecks base on latency tolerance, while SAM cannot distinguish tasks that are more sensitive to communication from others that are capable of absorbing the latency. Using application phase detection and accurate metrics that are deduced in these phases, SAM-MPH is able to understand the performance impact of communication due to data sharing.

Second, SAM-MPH identifies potential slowdown when putting two tasks on logical threads on the same physical core. Using this knowledge, it attempts to pair up applications such that they benefit from being placed on logical threads. If that is not possible, it attempts to schedule tasks so that they do not contend highly for the processor pipeline.

SAM-MPH's ability to identify data sharing and its impact on performance is the primary reason for the observed speedups in workloads 1–11. In each of these workloads, all applications exhibit data sharing. It is not possible to schedule all tasks such that tasks of an application remain inside the socket. SAM-MPH is able to prioritize applications that are more sensitive to the latency of communication due to data sharing.

In these workloads, ALS and PMF are the applications that are given the least priority and hence spread out across sockets. As discussed previously, these applications exhibit high ILP that is able to absorb the data communication latency. In addition to leveraging the tasks' ability to hide latency, SAM-MPH also identifies that ALS and PMF contend for the processor pipeline and avoids pairing their tasks together on the same physical core. Instead, SAM-MPH pairs each of their tasks with a task from the other applications to minimize resource contention. It is the combination of the these optimizations that yield consistent increase in performance of over 26% for these workloads.

SAM-M, being able to prioritize applications, can perform better than SAM. For workloads 1–11, SAM-M improves performance over Linux and SAM by 21% and 5% respectively. However, SAM-MP is faster than Linux and SAM by 25% and 8% respectively, demonstrating that SAM-MP's robustness adds value. SAM-MPH additionally mitigates contention on hyperthreads (due to ALS, RBM, and PMF), and is able to improve over SAM-MP to achieve performance closest to the standalone static schedule. SAM-MPH outperforms Linux and SAM by 30% and 13%.

For workloads 12–18, SAM-MPH identifies two applications with data sharing. The ideal decision for these workloads is to pin one application on each socket to localize all communication within a socket, which SAM-MPH and its variants correctly arrive at. SAM performs significantly slower since it does not have the notion of task groups and therefore is not able to separate the tasks. SAM relies on iteratively moving tasks onto the same socket since successful migrations will not cause additional inter-socket communication. Though this method works well in comparison with Linux, it does not achieve runtimes close to the optimal static runtime.

Workloads 19–21 contain applications with data sharing running simultaneously with other memory and CPU

Multiprog. workload #	Application mixes
1	12 ALS, 14 SGD, 14 LSGD
2	12 ALS, 14 SGD, 14 BSGD
3	12 ALS, 14 BSGD, 14 LSGD
4	12 ALS, 14 SVD, 14 BSGD
5	12 ALS, 14 SVD, 14 LSGD
6	12 ALS, 14 SVD, 14 SGD
7	12 ALS, 14 SVD, 14 RBM
8	12 ALS, 14 SGD, 14 RBM
9	12 PMF, 14 SGD, 14 RBM
10	12 PMF, 14 SGD, 14 BSGD
11	12 PMF, 14 SGD, 14 LSGD
12	20 SGD, 20 BSGD
13	20 SGD, 20 LSGD
14	20 SGD, 20 SVD
15	20 BSGD, 20 LSGD
16	20 LSGD, 20 ALS
17	20 LSGD, 20 SVD
18	20 BSGD, 20 SVD
19	6 SGD, 6 BSGD, 4 Mem, 4 CPU
20	6 BSGD, 6 LSGD, 4 Mem, 4 CPU
21	6 SGD, 6 LSGD, 4 Mem, 4 CPU
22	10 SGD, 10 BSGD
23	10 SGD, 10 LSGD
24	10 LSGD, 10 BSGD
25	10 LSGD, 10 ALS
26	10 SVD, 10 SGD
27	10 SVD, 10 BSGD
28	10 SVD, 10 LSGD
29	8 SVD, 8 LSGD
30	6 SVD, 6 LSGD

Table 2: Multiprogrammed application mixes. For each mix, the number preceding the application's name indicates the number of tasks it spawns. We use several combinations of applications to evaluate scenarios with varying data sharing and memory utilization.

bound tasks. These cases demonstrate the capability to balance load and resource utilization alongside reducing latency due to communication. In these cases, SAM-MPH and its variants achieve close to the standalone performance. SAM underperforms due to its inability to form groups between the two data sharing applications. It does, however, balance load and resource utilization. Workloads 23–30 also exhibit data sharing characteristics but use only 20 out of the 40 hardware contexts that are available. In these cases, SAM attempts to colocate all threads onto the same socket. This eliminates all inter-socket coherence traffic but increases pressure and contention for the last-level cache. Since SAM-MPH and its variants identify task grouping, they separate the two groups on the two available sockets, further reducing contention and eliminating communication due to data sharing.

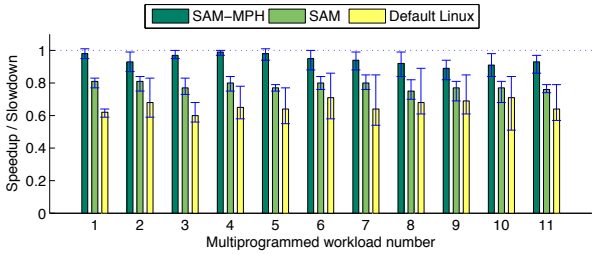


Figure 6: Performance of multiprogrammed workloads using SAM-MPH, SAM, and default Linux on the 80-CPU Haswell. The performance metric is the geometric mean of the individual application speedup (higher is better) compared to execution time obtained for a standalone run using the best static task→CPU mapping determined through offline testing. Whiskers represent the max-min speedup range for the individual applications within each workload.

Figure 6 shows the results obtained for the workload mixes listed in Table 3 on the four-socket 80-CPU Haswell. The workload mixes test SAM-MPH's ability to identify phases in applications that are most sensitive to data sharing. It also examines how SAM-MPH scales to a bigger machine with twice the number of processors and cores. We can see that SAM-MPH is able to achieve significantly better performance for all the workload mixes.

On average, we observe a 21% improvement over SAM and 43% improvement over stock Linux. While a reduction in performance compared to standalone execution is inevitable due to resource contention in multiprogrammed workloads, SAM-MPH is able to reduce the penalty. Performance improvement over Linux can be as high as 61% for our multiprogrammed workloads, while the minimum improvement was at 29% for workload 10.

Equally important, as the whisker plots in Figure 6 showing the minimum and maximum speedups for the individual applications in each workload show for the 4-socket 80-CPU Haswell machine, SAM-MPH reduces performance disparity (a measure of fairness) across applications in a workload in comparison to default Linux. The geometric mean of the minimum application speedup across all workload mixes is 0.889, 0.734, and 0.571 for SAM-MPH, SAM, and default Linux respectively. The corresponding values for the maximum speedup are 0.989, 0.822, and 0.795. On the 2-socket 40-CPU IvyBridge machine, the geometric mean of the minimum speedup is 0.953, 0.860, and 0.710, and that of the maximum is 1.0003, 0.932, and 0.839 respectively. Both SAM and SAM-MPH show a compressed spread.

5.4 Sensitivity Analysis

SAM-MPH relies on parameter thresholds to identify bottlenecks. In addition to the thresholds used in SAM,

Multiprog. workload #	Tasks per app	Application mixes
1	20	SGD, BSGD, SVD
2	20	SGD, BSGD, SVD, ALS
3	20	SGD, BSGD, SVD, LSGD
4	20	SGD, BSGD, RBM, LSGD
5	20	SGD, BSGD, RBM, SVD
6	20	SGD, BSGD, RBM, ALS
7	16	SGD, SVD, ALS, LSGD, BSGD
8	16	SGD, SVD, PMF, BGSD, LSGD
9	16	RBM, LSGD, SVD, PMF, BSGD
10	16	RBM, LSGD, SVD, PMF, ALS
11	16	SVD, SGD, RBM, BSGD, LSGD

Table 3: Multiprogrammed application mixes for experiments on the 80-CPU Haswell.

we use SPC and IPC values to prioritize task colocation based on latency tolerance and contention for pipeline resources. In this section, we look at sensitivity of SAM-MPH's behavior to these parameter thresholds. We increase/decrease the thresholds in steps of 5% to analyze the sensitivity of performance to these thresholds.

IPC is used to decide if tasks can be colocated on the same physical core. If the IPC threshold is too high, it is possible to map two compute intensive tasks on the same physical core, thereby slowing both down. A threshold increase of 20% (new IPC threshold of 1.08) can result in a performance reduction of about 7% on PMF. If the IPC threshold is too low, the mapper can miss a potential window to improve performance by colocating tasks that share data on the same physical core, thereby improving their performance and reducing contention by eliminating traffic on intra- and inter-socket interconnects. In our experiments, lowering the IPC threshold by 30% (new IPC threshold of 0.63) results in a loss in performance of 18% for the SGD application with 20 threads. The reduction in threshold created a false need to spread tasks across sockets in order to avoid colocating them on the same physical core, resulting in the slowdown.

SPC is used to prioritize applications when they are observed to have high coherence activity. Since SPC is approximated by attributing stalls due to all cache misses to coherence activity, SPC is reliable only at higher coherence activity levels. The coherence activity threshold used to identify when SPC is reliable is important to performance. We find a performance reduction of over 15% for mixed workload 8 when the threshold was increased by 30% (from 0.78×10^{-3} to 1.01×10^{-3}) due to missed opportunities. When the threshold is reduced by 50% (from 0.78×10^{-3} to 0.39×10^{-3}), we lose over 30% performance for workload 1 due to improper prioritization of applications.

Overall, we observe that while the value of the parameters is important to performance, SAM-MPH shows stable behavior over a reasonably broad range of values for these important parameters. In fact, we used the same thresholds, scaled for frequency, on the two platforms.

5.5 Overhead Assessment

SAM-MPH functionality can be divided into three distinct parts. The implementation complexity of each of these dictate the overall overhead of SAM-MPH. First, performance counters are read every 1 mSec. Second, every 100 mSecs, performance counter data is consolidated: application and socket-level bottlenecks are identified to be used to map tasks to cores. Finally, task mapping decisions are taken in order to improve performance. In order to measure the overhead of SAM-MPH, we perform a piecewise estimation since the implementation overhead is well within measurement error.

Reading performance counters are done at intervals of 1 mSec and consume 8.89 μSecs per call. This overhead is constant and does not vary with the number of processors/active tasks. Data consolidation, performed every 100 mSecs can consume a varying amount of time, primarily depending on the number of active applications. Worst case time consumption per SAM-MPH mapping call, including decision making and thread migration is 230 uSecs. Worst case behavior can be observed when each active task is its own application. The same overhead for when all cores are utilized but by only one application is about 14 uSecs. The additional overhead of over 200 uSecs is added by code that groups tasks into applications using address space information. The more distinct applications, the more the time spent on attributing tasks to applications. In most practical situations however, the number of applications will be significantly fewer than the number of cores. Overall, SAM-MPH adds a worst case overhead of just over 1%, which is far outweighed by its benefits.

SAM-MPH's data consolidation and decision making is implemented in a centralized fashion using a daemon process. On our machines, with 40 and 80 hardware threads, this implementation methodology works well. In the future, if SAM-MPH's overheads become a limitation, a distributed implementation may be warranted.

6 Related Work

Multicore resource contention and interference (particularly the shared last-level cache, off-chip bandwidth, and memory) has been well studied in previous work. Suh et al. [23] focus on minimizing cache misses using hardware counter-assisted marginal gain analysis. Page coloring [7, 26] has been used in the operating system memory allocator to effectively partition cache space without the need for specialized hardware features. Inter-task interference at the DRAM memory level has been

mitigated using parallelism-aware batch scheduling [18]. In an offline approach, Mars et al. [17] designed co-running microbenchmarks to control pressure on shared resources, and thereby predict the performance interference between colocated applications. While these techniques help manage resource contention, they do not address the impact of non-uniform topology on traffic due to data sharing.

ESTIMA [6] uses stall cycles to learn and predict application scalability on larger core counts using an offline approach. In contrast, our focus is on online multiprogrammed workload scheduling. Rao et al. [20] discuss using processor uncore pressure to minimize NUMA induced bottlenecks when scheduling virtual machines. While their approach of minimizing overall uncore pressure works to mitigate resource contention, it is not effective in eliminating resource pressure caused by data sharing.

Several efforts have also been made to automatically determine sharing among tasks. Tam et al. [24] utilized address sampling (available on Power processors) to identify task groups with strong data sharing. Tang et al. [25] relied on the number of accesses to "shared" cache lines to identify intra-application data sharing. Our past work [22] monitored and separated inter-CPU coherence activity from memory traffic to determine the benefits of consolidating tasks on the same socket versus distributing tasks across CPU sockets. Our work in this paper makes two new contributions. First, we identify latency tolerance in some workloads where inter-CPU coherence activity does not necessarily lead to CPU stalls and performance degradation. Second, we identify when the benefits of consolidating tasks on the same physical core due to data sharing outweigh performance loss due to contention for functional units and cache space.

Scheduling for simultaneous hardware multithreading, e.g., Intel's hyperthreads [13], has not been ignored in the past. Early work by Nakajima and Pallipadi [19] proposed two simple scheduling heuristics—1) task cache affinity to one hyperthread infers affinity to its sibling hyperthread; 2) scheduler should prefer a CPU whose sibling hyperthread is idle. Bulpin and Pratt [5] calibrated a blackbox linear model that predicts hyperthreading performance impact on a range of processor metrics. Their blackbox model provides no semantics on the hypothetical linear relationship and it is unclear how it applies broadly to other processors. Work in this paper monitors the cache coherence traffic, resulting stalls, and instruction retirement rates to understand the inter-CPU data sharing and potential latency tolerance, and thereby inform hyperthread colocation decisions.

The scalability of multicore and hardware multithreading has also been an emphasis in software system designs. For instance, Zhang et al. [27] presented user-space techniques (in the OpenMP runtime) to optimize inter-hyperthread data locality, instruction mix, and load balance. Multicore operating systems like Corey [4] and Multikernel [2] are designed to minimize cross-CPU sharing and synchronization for enhanced scalability. More recently, Callisto [9] is an OpenMP runtime system to handle synchronization and balance load on multicores. These efforts to improve software scalability are complementary to our CPU scheduling work—e.g., reduced data sharing traffic in some software tasks presents more flexibility to the scheduler that must consider resource contention, data sharing, and load balancing issues among all system and application tasks.

7 Conclusions

This paper presents new advances in resolving the tension between data sharing and resource contention in multicore task to core mapping. We make three specific contributions. First, we demonstrate the importance of identifying application latency tolerance, in addition to capturing data sharing traffic [22, 24, 25], in determining the true benefits of application and thread colocation. Second, we recognize that core-level sharing must pay attention to resource contention between hardware threads [5, 19] and show that a combination of IPC and coherence activity thresholds can inform the performance tradeoffs of core sharing. Third, we build an adaptive CPU socket and core sharing scheduler, called SAM-MPH, that uses history to avoid ineffective migrations due to oscillatory or transient behavior.

We perform experiments with a broad range of applications including SPEC CPU2000 [1], the PARSEC parallel benchmark suite [3], and GraphLab [16] / GraphChi [14] graph processing applications. Evaluation on a dual-socket, 40-CPU IvyBridge machine shows that SAM-MPH is 25% faster than Linux for standalone applications. On a larger 80-CPU Haswell machine with 4 sockets, SAM-MPH can halve the runtime of standalone workloads and can improve performance over Linux by up to 61% for multiprogrammed workloads. While SAM-MPH relies on thresholds to identify resource bottlenecks, our results show that performance is not sensitive to precise threshold values. Finally, SAM-MPH's runtime overhead in performance counter collection, analysis, and decision making is $\sim$1%, making it suitable for production use.

Acknowledgments This work was supported in part by the U.S. National Science Foundation grants CNS-1217372, CCF-1217920, CNS-1239423, CCF-1255729, CNS-1319353, CNS-1319417, and CCF-137224. We also thank the anonymous USENIX ATC reviewers and our shepherd Andy Tucker for comments that helped improve this paper.

References

[1] SPECCPU2006 benchmark. www.spec.org.

[2] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *22nd ACM Symp. on Operating Systems Principles (SOSP)*, 2009.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *8th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[5] J. R. Bulpin and I. A. Pratt. Hyper-threading aware process scheduling heuristics. In *USENIX Annual Technical Conf.*, pages 399–402, Anaheim, CA, Apr. 2005.

[6] G. Chatzopoulos, A. Dragojević, and R. Guerraoui. Estima: Extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 27:1–27:11, New York, NY, USA, 2016. ACM.

[7] S. Cho and L. Jin. Managing distributed, shared L2 caches through OS-level page allocation. In *39th Int'l Symp. on Microarchitecture (MICRO)*, pages 455–468, Orlando, FL, Dec. 2006.

[8] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.

[9] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *9th EuroSys Conf.*, Amsterdam, Netherlands, Apr. 2014.

[10] G. E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade - Second Edition*, pages 599–619. 2012.

[11] Y. Koren. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *14th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (SIGKDD)*, pages 426–434, Las Vegas, NV, 2008.

[12] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, Aug. 2009.

[13] D. Koufaty and D. T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, Apr. 2003.

[14] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-scale graph computation on just a PC. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 31–46, Hollywood, CA, 2012.

[15] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *USENIX Annual Technical Conf.*, Santa Clara, CA, July 2015.

[16] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Catalina Island, CA, July 2010.

[17] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible colocations. In *44th Int'l Symp. on Microarchitecture (MICRO)*, Porto Alegre, Brazil, Dec. 2011.

[18] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *35th Int'l Symp. on Computer Architecture (ISCA)*, pages 63–74, Beijing, China, June 2008.

[19] J. Nakajima and V. Pallipadi. Enhancements for hyper-threading technology in the operating system — seeking the optimal scheduling. In *Second Workshop on Industrial Experiences With Systems Software*, pages 25–38, Boston, MA, Dec. 2002.

[20] J. Rao, K. Wang, X. Zhou, and C. Z. Xu. Optimizing virtual machine scheduling in numa multicore systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 306–317, Feb 2013.

[21] R. Salakhutdinov and A. Mnih. Bayesian probabilistic matrix factorization using Markov Chain Monte Carlo. In *25th Int'l Conf. on Machine Learning (ICML)*, pages 880–887, Helsinki, Finland, 2008.

[22] S. Srikanthan, S. Dwarkadas, and K. Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In

USENIX Annual Technical Conf., Santa Clara, CA, July 2015.

[23] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, Apr. 2004.

[24] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Second EuroSys Conf.*, pages 47–58, Lisbon, Portugal, Mar. 2007.

[25] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 283–294, San Jose, CA, June 2011.

[26] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multi-core cache management. In *4th EuroSys Conf.*, pages 89–102, Nuremberg, Germany, Apr. 2009.

[27] Y. Zhang, M. Burcea, V. Cheng, R. Ho, and M. Voss. An adaptive OpenMP loop scheduler for hyperthreaded SMPs. In *17th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 256–263, San Francisco, CA, Sept. 2004.

[28] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix prize. In *Proc. 4th Intl Conf. Algorithmic Aspects in Information and Management, LNCS 5034*, pages 337–348. Springer, 2008.

Replex: A Scalable, Highly Available Multi-Index Data Store

Amy Tai[⋆†], Michael Wei[⋆‡], Michael J. Freedman[†], Ittai Abraham[⋆], Dahlia Malkhi[⋆]

[⋆]*VMWare Research,* [†]*Princeton University,* [‡]*University of California, San Diego*

Abstract

The need for scalable, high-performance datastores has led to the development of NoSQL databases, which achieve scalability by partitioning data over a single key. However, programmers often need to query data with other keys, which data stores provide by either querying every partition, eliminating the benefits of partitioning, or replicating additional indexes, wasting the benefits of data replication.

In this paper, we show there is no need to compromise scalability for functionality. We present Replex, a datastore that enables efficient querying on multiple keys by rethinking data placement during replication. Traditionally, a data store is first globally partitioned, then each partition is replicated identically to multiple nodes. Instead, Replex relies on a novel replication unit, termed *replex*, which partitions a full copy of the data based on its unique key. Replexes eliminate any additional overhead to maintaining indices, at the cost of increasing recovery complexity. To address this issue, we also introduce *hybrid replexes*, which enable a rich design space for trading off steady-state performance with faster recovery. We build, parameterize, and evaluate Replex on multiple dimensions and find that Replex surpasses the steady-state and failure recovery performance of HyperDex, a state-of-the-art multi-key data store.

1 Introduction

Applications have traditionally stored data in SQL databases, which provide programmers with an efficient and convenient query language to retrieve data. However, as storage needs of applications grew, programmers began shifting towards NoSQL databases, which achieve scalability by supporting a much simpler query model, typically by a single primary key. This simplification made scaling NoSQL datastores easy: by using the key to divide the data into partitions or "shards", the datastore could be efficiently mapped onto multiple nodes. Unfortunately, this model is inconvenient for programmers, who often still need to query data by a value other than the primary key.

Several NoSQL datastores[1, 3, 14, 9, 15, 7] have emerged that can support queries on multiple keys through the use of secondary indexes. Many of these datastores simply query all partitions to search for an entry which matches a secondary key. In this approach, performance quickly degrades as the number of partitions increases, defeating the reason for partitioning for scalability. HyperDex [12], a NoSQL datastore which takes another approach, generates and partitions an additional copy of the datastore for each key. This design allows for quick, efficient queries on secondary keys, but at the expense of storage and performance overhead: supporting just one secondary key doubles storage requirements and write latencies.

In this paper, we describe Replex, a scalable, highly available multi-key datastore. In Replex, each full copy of the data may be partitioned by a different key, thereby retaining the ability to support queries against multiple keys without incurring a performance penalty or storage overhead beyond what is required to protect the database against failure. In fact, since Replex does not make unnecessary copies of data, it outperforms other NoSQL systems during both steady-state and recovery.

To address the challenge of determining when and where to replicate data, we explore, develop, and parameterize a new replication scheme, which makes use of a novel replication unit we call a *replex*. The key insight of a replex is to combine the need to replicate for fault-tolerance and the need to replicate for index availability. By merging these concerns, our protocol avoids using ex-

traneous copies as the means to enable queries by additional keys. However, this introduces a tradeoff between recovery time and storage cost, which we fully explore (§ 3). Replex actually recovers from failure faster than other NoSQL systems because of storage savings during replication.

We implement (§ 4) and evaluate (§ 5) the performance of Replex using several different parameters and consider both steady-state performance and performance under multiple failure scenarios. We compare Replex to Hyperdex and Cassandra and show that Replex's steady-state performance is 76% better than Hyperdex and on-par with Cassandra for writes. For reads, Replex outperforms Cassandra by as much as 2-9× while maintaining performance equivalent with HyperDex. In addition, we show that Replex can recover from one or two failures 2-3× faster than Hyperdex, all while using a fraction of the resources.

Our results contradict the popular belief that supporting multiple keys in a NoSQL datastore is expensive. With replexes, NoSQL datastores can easily support multiple keys with little overhead.

2 System Design

We present Replex's data model and replication design, which enables fast index reads and updates while being parsimonious with storage usage.

2.1 Data Model and API

Replex stores data in the form of RDBMS-style tables: every table has a schema that specifies a fixed set of columns, and data is inserted and replicated at the row-granularity. Every table also specifies a single column to be the primary key, which becomes the default index for the table.

As with traditional RDBMSs, the user can also specify any number of additional indexes. An index is defined by the set of columns that comprise the index's **sorting key**. For example, the sorting key for the primary index is the column of the primary key.

The client queries we focus on in this paper are $insert(r)$, where r is a row of values, and $lookup(R)$, where R is a row of predicates. Predicates can be null, which matches on anything. Then $lookup(R)$ returns all rows r that match on all predicates in R. The non-null predicates should correspond to the sorting key of an index in the table. Then that index is used to find all matching rows.

Henceforth, we will refer to the data stored in Replex as the *table*. Then Replex is concerned with maintaining the indexes of and replicating the table.

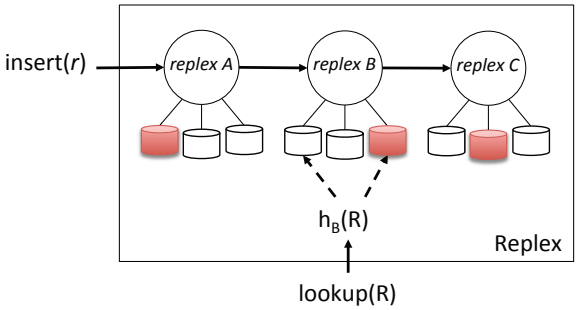


Figure 1: Every replex stores the table across of a number of partitions. This diagram shows the system model for a table with 3 indexes. When a row r is inserted, h_A, h_B, and h_C determine which partition (shaded) in the replex stores r. Similarly, a lookup on a replex is broadcast to a number of partitions based on h.

2.2 Data Partitioning with Replexes

In order to enable fast queries by a particular index, a table must be partitioned by that index. To solve this problem, Replex builds what we call a *replex* for every index. A replex stores a table and shards the rows across multiple partitions. All replexes store the same *data* (every row in the table), the only difference across replexes is the way data is partitioned and sorted, which is by the sorting key of the index associated with the replex.

Each replex is associated with a sharding function, h, such that $h(r)$ defines the partition number in the replex that stores row r. For predicate R, $h(R)$ returns a set because the rows of values that satisfy R may lie in multiple partitions. The only columns that affect h are the columns in the sorting key of the index associated with the replex.

A novel contribution of Replex is to treat each partition of a replex as first-class replicas in the system. Systems typically replicate a row for durability and availability by writing it to a number of replicas. Similarly, Replex uses chain replication [27] to replicate a row to a number of replex partitions, each of which sorts the row by the replex's corresponding index, as shown in Figure 1; in Section 2.3 we explain why we choose chain replication. The key observation is that after replication, Replex has both replicated *and* indexed a row. There is no need for explicit indexing.

By treating replexes as true replicas, we eliminate the overheads associated with maintaining and replicating

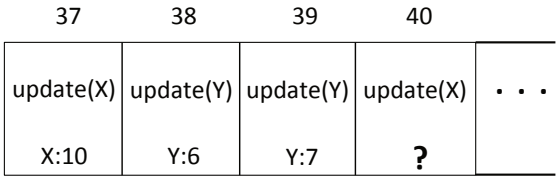


Figure 2: Consider storing every log entry in a Replex table. For linearizability, a local timestamp cannot appear to go backwards with respect to the global timestamp. For example, tagging in last entry with local timestamp X:9 violates the semantics of the global timestamp.

individual index structures, which translates to reductions in network traffic, operation latency, and storage inflation.

2.3 Replication Protocol

Replacing replicas with replexes requires a modified replication protocol. The difficulty arises because individual replexes can have requirements, such as uniqueness constraints, that cause the same operation to be both valid and invalid depending on the replex. Hence before an operation can be replicated, a consensus decision must be made among the replexes to agree on the validity of an operation.

As an example of an ordering constraint, consider a distributed log that totally orders updates to a number of shared data structures, *a la* state machine replication. In addition to the global ordering, each data structure requires a local ordering that must reflect the global total ordering. For example, suppose there are two data structures X and Y, and a subset of the log is shown in Figure 2. To store the updates in Replex, we can create a table with two columns: a global timestamp and a local timestamp. Because consumers of the log will want to look up entries both against the global timestamp and within the sublog of a specific data structure, we also specify an index per column; examples of logs with such requirements appear in systems such as Corfu [4], Hyder [6], and CalvinFS [24].

Then the validity requirement in this case is a dense prefix of timestamps: a timestamp t cannot be written until all timestamps $t' < t$ have been inserted into the table; this is true for both the local and global timestamps. For example, an attempt to insert the row (40, X:9) would be valid by the index of the global timestamp, but invalid by the index of the local timestamp, because the existence of X:10 in the index means X:9 must have already been inserted. Then the row should not be inserted into the table; this is problematic if the first replex has already processed the insert, which means lookups on the first index will see row (40, X:9).

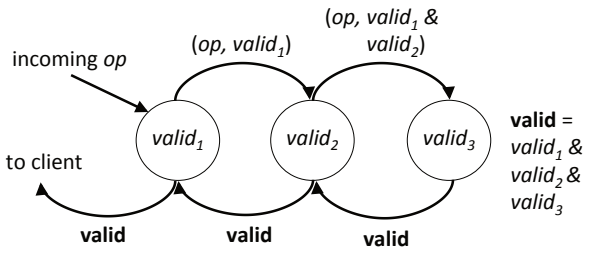


Figure 3: Each node represents an index. This modified replication protocol has two phases: 1) Top phase: propagates the operation to all relevant partitions and collects each partition's decision. 2) Bottom phase: the last partition aggregates these decisions into the final **valid** boolean, which is then propagated back up the chain. When a replex receives **valid**, it knows to commit or abort the operation

Datastores without global secondary indexes do not have this validity problem, because a key is only sorted by a single index. Datastores with global secondary indexes employ a distributed transaction for update operations, because an operation must be atomically replicated as valid or invalid across all the indexes [11]. Because replexes are similar to global secondary indexes, a distributed transaction can do the job. But to use a distributed transaction for every update operation would cripple system throughput.

To remove the need for a distributed transaction in our replication protocol, we modify chain replication to include a consensus protocol. We choose chain replication instead of quorum-based replication because all replexes must participate to determine validity. As in chain replication, our protocol visits every replex in some fixed order. Figure 3 illustrates the steps in this new replication protocol.

Our new protocol can be split into two phases: (1) **consensus phase**, where we propagate the operation to all replexes, as in chain replication. The actual partition within the replex that handles the operation is the partition that will eventually replicate the operation, as depicted in Figure 1. As the protocol leaves each partition, it collects that partition's validity decision. When this phase reaches the last partition in the chain, the last partition aggregates each partition's decision into a final decision, which is simply the logical AND of all decisions: if there is a single abort decision, the operation is

invalid. (2) **replication phase**, where the last partition initiates the propagation of this final decision back up the chain. As each partition receives this final decision, if the decision is to abort, then the partition discards that operation. If the decision is to commit, then that partition commits the operation to disk and continues propagating the decision.

It is guaranteed that when the client sees the result of the operation, all partitions will agree on the outcome of the operation, and if the operation is valid, all partitions will have made the decision durable. An intuitive proof of correctness for this consensus protocol is simple. We can treat the first phase of our protocol as an instance of chain replication, which is an instance of Vertical Paxos, which has existing correctness proofs [16]. The second phase of our protocol is simply a discovery phase in Paxos protocols and is hence irrelevant in the proof of correctness. This discovery phase is necessary for replexes to discover the final decision so they may persist (replicate) necessary data, but has no bearings on the consensus decision itself.

It is possible for a client to see committed operations at one replex before another. For example, suppose client 1 is propagating an operation to replexes A and B. The operation reaches B and commits successfully, writing the commit bit at B. Then this committed operation is visible to client 2 that queries replex B, even though client 2 cannot see it by querying replex A, if the commit bit is still in flight. Note that this does not violate the consensus guarantee, because any operation viewed by one client is necessarily committed.

Our protocol is similar to the CRAQ protocol which adds dirty-read bits to objects replicated with chain replication [23]. The difference between the two protocols is that CRAQ operates on objects, rather than operations: our protocol determines whether or not an operation may be committed to an object's replicated state machine history, while CRAQ determines whether or not an object is dirty. In particular, operations can be aborted through our protocol.

Finally, we observe that our replication protocol does not allow writes during failure. In chain replication, writes to an object on a failed node cannot resume until its full persisted history has been restored; similarly, writes may not be committed in Replex until the failed node is fully recovered.

2.4 Failure Amplification

Indexing during replication enables Replex to achieve fast steady-state requests. But there is a cost, which becomes evident when we consider partition failures.

Failed partitions bring up two concerns: how to reconstruct the failed partition and how to respond to queries that would have been serviced by the failed partition. Both of these problems can be solved as long as the system knows how to find data stored on the failed partition. The problem is even though two replexes contain the same *data*, they have different sharding functions, so replicated data is scattered differently.

We define **failure amplification** as the overhead of finding data when the desired partition is unavailable. We characterize failure amplification along two axes: 1) disk IOPS and CPU: the overhead of searching through a partition that is sorted differently, 2) network traffic: the overhead of broadcasting the read to all partitions in another replex. For the remainder of the paper, we use failure amplification to compare recovery scenarios.

For example, suppose a user specifies two indexes on a table, which would be implemented as two replexes in Replex. If a partition fails, a simple recovery protocol would redirect queries originally destined for the failed partition to the other replex. Then the failure amplification is maximal: the read must now be broadcast to every partition in the other replex, and at each partition, a read becomes a brute-force search that must iterate through the entire local storage of a partition.

On the other hand, to avoid failure amplification within a failure threshold f, one could introduce f replexes with the same sharding function, h; these are the exact replicas of traditional replication. There is no failure amplification within the failure threshold, because sharding is identical across exact replicas; the cost is storage and network overhead in the steady-state.

The goal is to capture the possible deployments in between these two extremes. Unfortunately, without additional machinery, this space can only be explored in a discrete manner: by adding or removing exact replicas. In the next section, we introduce a construct that allows fine-grained reasoning within this tradeoff space.

3 Hybrid Replexes

Suppose a user schema specifies a single table with two indexes, A and B, so Replex builds two replexes. As mentioned before, as soon as a partition in either replex fails, reads to that partition must now visit all partitions in the other replex, the disjoint union of which is the entire dataset.

One strategy is to add replexes that are exact replicas. For example, we can replicate replex A, as shown in Figure 4. Then after one failure, reads to replex A do not see any failure amplifcation. However, adding another

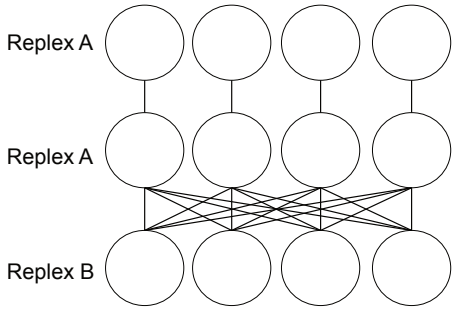


Figure 4: In graph depictions of replexes, nodes are partitions and edges indicate two partitions might share data. For example, because replexes *A* and *B* have independent sharding functions, it is possible for all combinations of nodes to share data. This graph shows a simple solution to reduce the failure amplification experienced by replex *A*, which is to replicate *A* again.

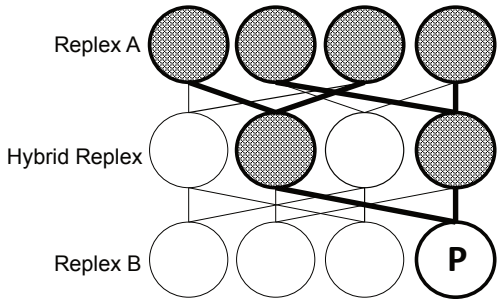


Figure 5: Each node is connected to exactly 2 nodes in another replex. This means that partitions in both replexes will see only 2x failure amplification after a single failure.

copy of replex *A* does not improve failure amplification for reads to *B*: if a partition fails in replex *B*, failure amplification still becomes worst-case.

To eliminate failure amplification of a single failure on both replexes, the user must create exact replicas of both replexes, thereby doubling all storage and network overheads previously mentioned.

Instead, we present **hybrid replexes**, which is a core contribution of Replex. The basic idea behind hybrid replexes is to introduce a replex into the system that increases failure resilience of *any number* of replexes; an exact replica only increases failure resilience of a single replex. We call them hybrid replexes because they enable a middleground between adding either one or zero exact-copy replexes.

A hybrid replex is shared by replex *A* if h_{hybrid} is dependent on h_A. In the next few sections, we will explain how to define h_{hybrid} given the shared replexes.

Hybrid replexes are a building block for constructing a system with more complex failure amplification models per replex. To start with, we show how to construct a hybrid replex that is shared across two replexes.

3.1 2-Sharing

Consider replexes *A* and *B* from before. The system constructs a new, *hybrid* replex that is shared by *A* and *B*. Assume that all replexes have 4 partitions; in Section 3.2 we will consider *p* partitions.

To define the hybrid replex, we must define h_{hybrid}. Assume that each partition in each replex in Figure 5 is numbered from left to right from 0-3. Then:

$$h_{hybrid}(r) = 2 \cdot (h_A(r) \pmod 2) + h_B(r) \pmod 2 \quad (1)$$

The graph in Figure 5 visualizes h_{hybrid}. The partition in the hybrid replex that stores row *r* is the partition connecting the partition in *A* and the partition in *B* that store *r*. Edges indicate which partitions in another replex share data with a given partition; in fact, if there exists a path between any two partitions, then those two partitions share data. Then any read that would have gone to a failed node can equally be serviced by visiting all partitions in an replex that are path-connected to the failed node.

For example, *P* shares data with exactly two partitions in the hybrid replex, and all four partitions in replex *A*. This means that when *P* fails, reads can either go to these two partitions in the hybrid replex or all four partitions in replex *A*, thereby experiencing 2x or 4x failure amplification, respectively. Then it is clear that reads should be redirected to the hybrid replex. Furthermore, because the hybrid replex overlaps attributes with replex *B*, any read redirected to the hybrid replex can be faster compared to a read that is redirected to replex *A*, which shares no attributes with replex *B*.

Figure 5 helps visualize how a partition in *any* replex will only cause failure amplification of two: each partition has an outcast of two to adjacent replexes. Hence by adding a single replex, we have reduced the failure amplification for *all* replexes after one failure. Contrast this with the extra replica approach: if we only add a single exact replica of replex *A*, replex *B* would still experience 4x failure amplification after a single failure.

This hybrid technique might evoke erasure coding in the reader. However, as we explain in Section 6, erasure

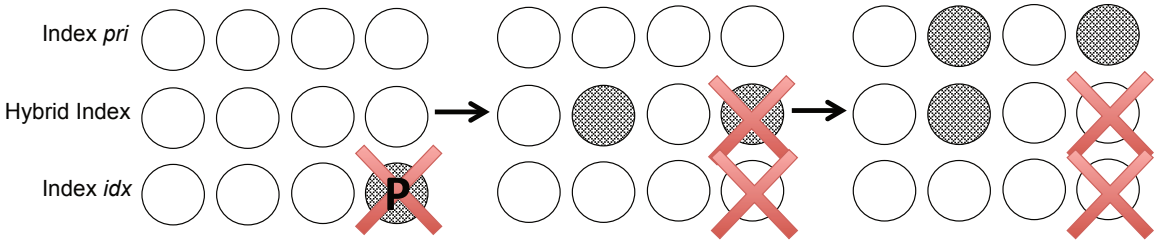


Figure 6: Graceful degradation. Shaded nodes indicate the nodes that must be contacted to satisfy queries that would have gone to partition P. As failures occur, Replex looks up replacement partitions for the failed node and modifies reads accordingly. Instead of contacting an entire replex after two failures, reads only need to contact a subset.

coding solves a different problem. In erasure coding, parity bits are scattered across a cluster in *known* locations. The metric for the cost of a code is the reconstruction overhead after collecting all the parity bits. On the other hand, with replexes, there is no reconstruction cost, because replexes store full rows. Instead, hybrid replexes address the problem of *finding* data that is sharded by a different key in a different replex.

Hybrid replexes also smooth out the increase in failure amplification as failures occur. The hybrid approach introduces a recursive property that enables graceful read degradation as failures occur, as shown in Figure 6.

In Figure 6, reads to P are redirected as cascading failures happen. When P fails, the next smallest set of partitions— those in the hybrid replex— are used. If a partition in *this* replex fails, then the system replaces it in a similar manner. Then the full set of partitions that must be accessed is the three shaded nodes in the rightmost panel. Three nodes must fail concurrently before the worst set, all partitions in an replex, is used. The system is only fully unavailable for a particular read if after recursively expanding out these partition sets it cannot find a set without a failed node.

This recursion stops suddenly in the case of exact replicas. Suppose a user increases the failure resilience of A by creating an exact replica. As the first failure in A occurs, the system can simply point to the exact replica. When the second failure happens, however, reads are necessarily redirected to all partitions in B.

3.2 Generalizing 2-sharing

In general, we can parametrize a hybrid replex by n_1 and n_2, where $n_1 \cdot n_2 = p$ and p is the number of partitions per replex. Then:

$$h_{hybrid}(r) = n_2 \cdot (f_A(r) \pmod{n_1}) + f_B(r) \pmod{n_2} \quad (2)$$

Applying this to Figure 5, each partition in A would have an outcast of n_2 instead of two, and each partition in B would have an incast of n_1. Then when partitions in replex A fail, reads will experience n_2-factor amplification, while reads to partitions in replex B will experience n_1-factor failure amplification. The intuition is to think of each partition in the hybrid replex as a pair: (x,y), where $0 \le x < n_1$ and $0 \le y < n_2$. Then when a partition in replex A fails, reads must visit all hybrid partitions $(x,*)$ and when a partition in replex B fails, reads must visit all hybrid partitions $(*,y)$. The crucial observation is that $n_1 \cdot n_2 = p$, so the hybrid layer enables only $n_1, n_2 = O(\sqrt{p})$ amplification of reads during failure, as opposed to $O(p)$.

n_1 and n_2 become tuning knobs for a hybrid replex. A user can assign n_1 and n_2 to different replexes based on importance. For example, if $p = 30$, then a user might assign $n_1 = 5$ and $n_2 = 6$ to two replexes A and B that are equally important. Alternatively, if the workload mostly hits A, which means failures in A will affect a larger percentage of queries, a user might assign $n_1 = 3$ and $n_2 = 10$. Even more extreme, the user could assign $n_1 = 1$ and $n_2 = 30$, which represents the case where the hybrid replex is an exact replica of replex A.

3.3 More Extensions

In this section, we discuss intuition for further generalizing hybrid replexes. Explicit construction requires defining complex h_{hybrid} that is beyond the scope of this paper.

Hybrid replexes can be shared across r replexes, not just two as presented in the previous sections. To decrease failure amplification across r replexes, we create a hybrid replex that is shared across these r replexes. To parametrize this space, we use the same notation used to generalize 2-sharing. In particular, think of each partition in the hybrid replex as an r-tuple: $(n_1, \ldots, n_r)$. Then when some partition in the qth replex fails, reads must visit all partitions $(*, \ldots, *, x_q, *, \ldots, *)$. Then failure amplification after one failure becomes $O(p^{\frac{r-1}{r}})$. As ex-

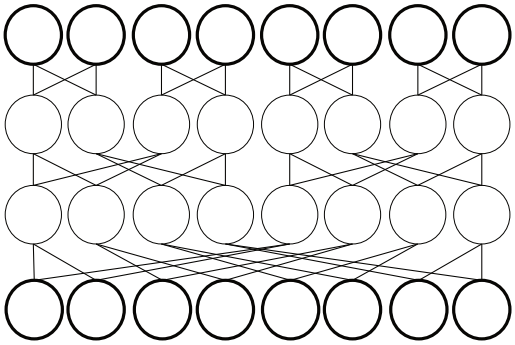


Figure 7: Inserting two hybrid replexes in between two replexes (in bold). Each node in the graph has outcast 2, which means after any partition fails, failure amplification will be at most 2x. After two failures, amplification will be 3x; after three, it will be 4x.

pected, if more replexes share a hybrid replex, improvement over $O(p)$ failure amplification becomes smaller.

For example, suppose a table requires 4 indexes, which will be translated into 4 replexes. Then a hybrid replex is not necessary for replication, but rather can be inserted at the discretion of the user, who might want to increase read availability during recovery. Simply paying the costs of an additional 4-shared hybrid replex can greatly increase failure read availability.

We can also increase the number of hybrid replexes inserted between two replexes. For example, we can insert *two* hybrid replexes between every two desired replexes, as shown in Figure 7. Then two hybrid replexes enable $O(p^{1/3})$ amplification of reads during failure, at the expense of introducing yet another replex. If two replexes share k hybrid replexes, then there will be $O(p^{\frac{1}{k+1}})$ amplification of reads during failure. As expected, if two replexes share more hybrid replexes, the failure amplification becomes smaller. Furthermore, Figure 7 shows that adding more hybrid replexes enables better cascading failure amplification. The power of hybrid replexes lies in tuning the system to expected failure models.

4 Implementation

We implemented Replex on top of HyperDex, which already has a framework for supporting multi-indexed data. However, we could have implemented replexes and hybrid replexes on any system that builds indexes for its data, including RDBMSs such as MySQL Cluster, as well as any NoSQL system. We added around 700 lines of code to HyperDex, around 500 of which were devoted to make data transfers during recovery performant.

HyperDex implements copies of the datastore as subspaces. Each subspace in HyperDex is associated with a hash function that shards data across that subspace's partitions. We replaced these subspaces with replexes, which can take an arbitrary sharding function. For example, in order to implement hybrid replexes, we initialize a generic replex and assign h to any of the h_{hybrid} discussed in Section 3. We reuse the chain replication that HyperDex provides to replicate across subspaces.

To satisfy a lookup query, Replex calculates which nodes are needed for lookup from the system configuration that is fetched from a coordinator node. A lookup is executed against any number of replexes, so Replex uses the sharding function of the respective replex to identify relevant partitions. The configuration tells Replex the current storage nodes and their status. We implemented the recursive lookup described in Section 3.1 that uses the configuration to find the smallest set that contains all available partitions. For example, if there are no failures, then the smallest set is the original partitions. Replex implements this lookup functionality in the client-side HyperDex library. The client then sends the search query to all nodes in the final set and aggregates the responses; the client library waits to hear from all nodes before returning.

This recursive construction is used again in Replex's recovery code. In order to reconstruct a partition, Replex calculates a minimal set of partitions to contact and sends each member a reconstruction request with a predicate. The predicate can be thought of as matching on $h(r)$, where h is the sharding function of the replex to which the receiving partition belongs. When a node receives the reconstruction request, it maps the predicate across its local rows and only sends back rows that satisfy the predicate.

Finally, to run Replex, we set HyperDex's fault tolerance to $f = 0$.

5 Evaluation

Our evaluation is driven by the following questions:

- How does Replex's design affect steady-state index performance? (§ 5.1)
- How do hybrid replexes enable superior recovery performance? (§ 5.2)
- How can generalized 2-sharing allow a user to tune failure performance? (§ 5.3)

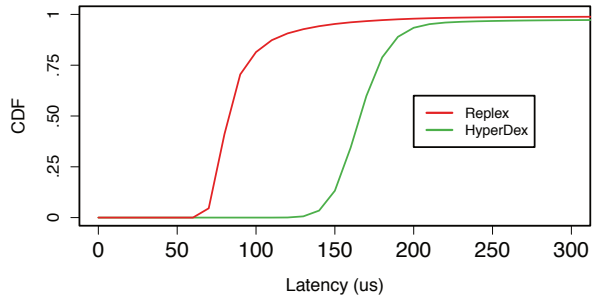


Figure 8: Insert latency microbenchmark CDF

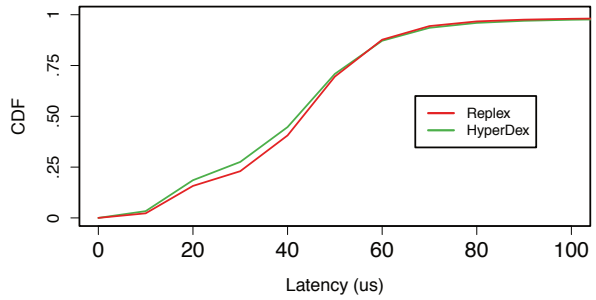


Figure 9: Read latency microbenchmark CDF

Name	Workload	Total Operations
Load	100% Insert	10 M
A	50% Read/50% Update	500 K
B	95% Read/5% Update	1 M
C	100% Read	1 M
D	95% Read/5% Insert	1 M
E	95% Scan/5% Insert	10 K
F	50% Read/50% Read-Modify	500 K

Table 1: YCSB workloads.

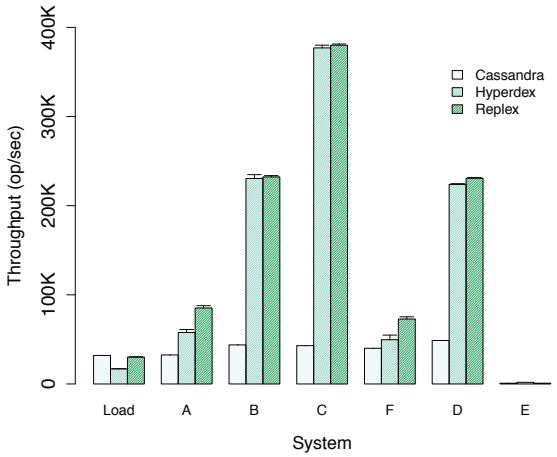


Figure 10: Mean throughput for full YCSB suite over 3 runs. Error bars indicate standard deviation. Results grouped by workload, in the order they are executed in the benchmark.

- How do hybrid replexes enable better resource tradeoffs with *r*-sharing? (§ 5.4)

Setup. All physical machines used had 8 CPUs and 16GB of RAM running Linux (3.10.0-327). All machines ran in the same rack, connected via 1Gbit links to a 1Gbit top-of-rack switch. 12 machines were designated as servers, 1 machine was a dedicated coordinator, and 4 machines were 64-thread clients. For each experiment, 1 or 2 additional machines were allocated as recovery servers.

5.1 Steady-State Performance

To analyze the impact of replacing replicas with replexes, we report operation latencies in Replex. We specify a table in Replex with two indexes: the primary index and a secondary index. We configure Replex to build a single hybrid replex, so Replex builds 3 full replexes during the benchmark; we call this system Replex-3 in Table 2. Because Replex-3 builds 3 replexes, data is tolerant to 2 failures. Hence, we also set HyperDex to three-way replicate data objects.

Read latency for Replex is identical to HyperDex's, because reads are simply done on the primary index of both systems; we report the read CDF in Figure 9. More importantly, the insert latency for Replex is consistently 2x less than the latency of a HyperDex insert, as in Figure 8. This is because one Replex insert visits 3 partitions while one HyperDex insert visits $2 \cdot 3 = 6$ partitions; these values are the replication factor denoted in Table 2. In fact, the more indexes a user builds, the larger the factor of difference in latency inserts. This helps to demonstrate Replex's scalability compared to HyperDex.

Figure 10 reports results from running a full YCSB benchmark on 3 systems: Cassandra, HyperDex, and Replex-3; Yahoo Cloud-Serving Benchmark (YCSB) is a well established benchmark for comparing NoSQL stores [10]. Because Replex-3 is tolerant to 2 failures, we also set Cassandra and HyperDex to three-way replicate data objects. In the load phase, YCSB inserts 10 million 100 byte rows into the datastore.

Replex-3's lower latency insert operations translate to higher throughput on the load portion and Workloads A, F than both HyperDex and Cassandra; these are the workloads with inserts/updates. Workload C has comparable performance to HyperDex, because these reads can be performed on the index that HyperDex builds. Cas-

System	Failures Tolerated	Replication Factor
Replex-2	1	2x
Replex-3	2	3x
HyperDex	2	6x

Table 2: Systems evaluated.

sandra has comparable load throughput because writes are replicated in the background; Cassandra writes return after visiting a single replica while our writes return after visiting all 3 replexes for full durabilty.

5.2 Failure Evaluation

In this section, we examine the throughput of three systems as failures occur: 1) HyperDex with two subspaces, 2) Replex with two replexes (Replex-2), and 3) Replex with two replexes and a hybrid replex (Replex-3). Each system has 12 virtual partitions per subspace or replex. One machine is reserved for reconstructing the failed node. Each system automatically assigns the 12 virtual partitions per replex across the 12 server machines.

For each system we specify a table with a primary and secondary index. We run two experiments, one that loads 1 million rows of size 1KB bytes and one that loads 10 million rows of size 100 bytes; the second experiment demonstrates recovery behavior when CPU is the bottleneck. We then start a microbenchmark where clients read as fast as possible against both indexes. Reads are split 50:50 between the two indexes. We kill a server after 25 seconds. Figure 11 shows the read throughput in the system as a function of time, and Tables 3 and 4 report average recovery statistics.

Recovery time in each system depends on the *size* of the data loss, which depends on how much data is stored on a physical node. The number of storage nodes is a constant across all three systems, so the amount of data stored on each node is proportional to the total amount of data across all replicas; recovery times in Tables 3 and 4 are approximately proportional to the Replication Factor column in Table 2. By replacing replicas with replexes, Replex can reduce recovery time by 2-3x, while also using a fraction of the storage resources.

Interestingly, Replex-2 recovers the fastest out of all systems, which suggests the basic Replex design has performance benefits even without adding hybrid replexes.

Recovery throughput shows one of the advantages of the hybrid replex design. In Table 3, Replex-2 has minimal throughput during recovery, because each read to the

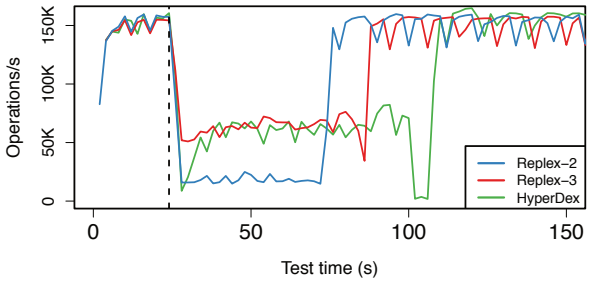


Figure 11: We crash a server at 25s and report read throughput for Replex-2, Replex-3, and Hyperdex. Systems are loaded with 10 million, 100 byte rows. All three systems experience a dip in throughput right before returning to full functionality due to the cost of reconfiguration synchronization, which introduces the reconstructed node back into the system configuration.

System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-2	50 ± 1	$18{,}989 \pm 1{,}883$
Replex-3	60 ± 1	$65{,}780 \pm 3{,}839$
HyperDex	105 ± 17	$34{,}697 \pm 19{,}003$

Table 3: Recovery statistics of one machine failure after 25 seconds. 10 million, 100 byte records. Results reported as average time $\pm$ standard deviation of 3 runs.

System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-2	6.7 ± 0.57	$70{,}084 \pm 5{,}980$
Replex-3	8.7 ± 0.56	$110{,}280 \pm 11{,}232$
HyperDex	20.0 ± 2.65	$127{,}232 \pm 85{,}932$

Table 4: Recovery statistics of one machine failure after 25 seconds. 1 million, 1KB records. Results reported as average time $\pm$ standard deviation of 3 runs.

failed node must be sent to all 12 partitions in the other replex. These same 12 partitions are also responsible for reconstructing the failed node; each of the partitions must iterate through their local storage to find data that belongs on the failed node. Finally, these 12 partitions are still trying to respond to reads against the primary index, hence system throughput is hijacked by reconstruction throughput and the amplified reads. Replex-2 throughput is not as bad in Table 4, because 1 million rows does not bottleneck the CPU during recovery.

The Replex-3 alleviates the stress of recovery by introducing the hybrid replex. First, each read is only amplified 3 times, because the grid constructed by the hybrid replex has dimensions $n_1 = 3, n_2 = 4$. Second, only 3 partitions are responsible for reconstructing the

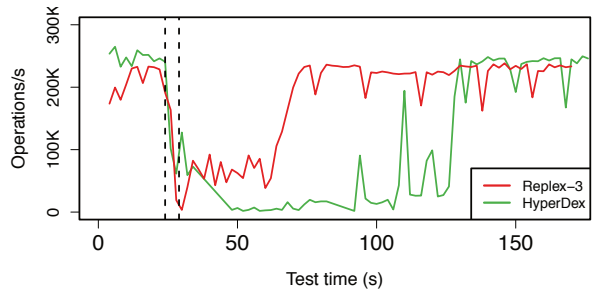


Figure 12: Read throughput after two failures. We crash one server at 25s and then a second at 30s. Request pile-up because throughput is used for recovery is responsible for the jumps in HyperDex throughput.

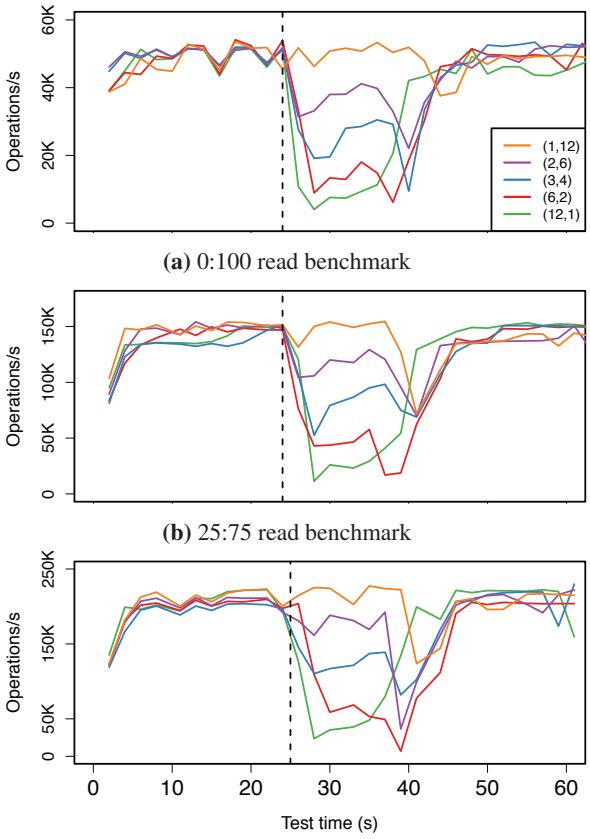
System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-3	37.6 ± 1.2	$60{,}844 \pm 27{,}492$
HyperDex	98.0 ± 11	$30{,}220 \pm 8{,}104$

Table 5: Recovery statistics of two machine failures at 25s and 30s. Results reported as average time $\pm$ standard deviation of 3 runs. Recovery time is measured from the first failure.

failed node. In fact, in both experiments, Replex-3 achieves recovery throughput comparable to that of HyperDex, which has no failure amplification, whilst adding little recovery time.

Finally, we highlight the hybrid replex design by running an experiment that causes two cascading failures. Replex-2 only tolerates two failures, so we do not include it in this experiment. Figure 15 shows the results when we run the same 50:50 read microbenchmark and crash a node at 25s and 30s. We reserve an additional 2 machines as spares for reconstruction. We run the experiment where each system is loaded with 1 million, 1K rows.

Figure 15 stresses the advantages of graceful degradation, enabled by the hybrid replex. We observe that experiencing two failures more than quadruples the recovery time in HyperDex. This is because the two reconstructions occur sequentially and independently. In Replex-3, failing a second partition causes reduced recovery throughput, because the second failed partition must rebuild from partitions that are actively serving reads. However, recovery time is bounded because reconstruction of the failed nodes occurs in parallel. When the second failed partition recovers, throughput nearly returns to normal.



(a) 0:100 read benchmark

(b) 25:75 read benchmark

(c) 50:50 read benchmark

Figure 13: We crash a machine at 25s. Each graph shows read throughput for Replex-3 with five different hybrid parametrizations and the labelled workload. Although $(12,1)$ has the worst throughput during failure, it recovers faster than the other parametrizations because recovery is spread across more partitions.

5.3 Parametrization of the Hybrid Replex

As discussed in Section 3.2, any hybrid replex $\mathscr{H}$ can be parametrized as (n_1, n_2). Consider the Replex-3 setup, which replicates operations to replexes in the order $A \rightarrow \mathscr{H} \rightarrow B$. If $\mathscr{H}$ is parametrized by (n_1, n_2), then failure of a partition in B will result in n_1-factor read amplification, and a failure in A will result in n_2-factor read amplification. In this section we investigate the effect of hybrid replex parameterization on throughput under failure.

We load each parametrization of Replex-3 with 1 million 1KB entries and fail a machine at 25s. Four separate client machines run an $a : b$ read benchmark, where a percent of reads go to replex A and b percent of reads go to replex B. Figure 13 shows the throughput results

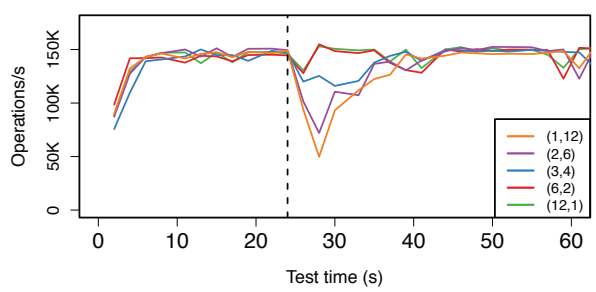


Figure 14: Replex-3 throughput with a 25:75 read benchmark. We crash a machine in replex A at 25s.

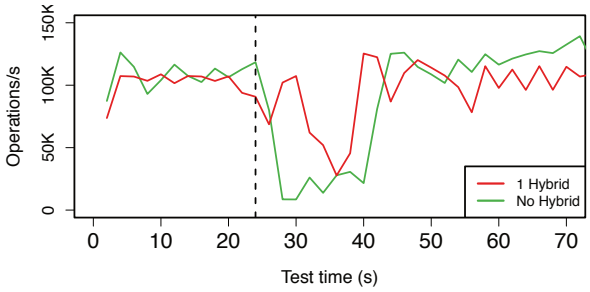


Figure 15: Read throughput after a failure at 25s with a 33:33:33 benchmark.

when a machine in B is killed at 25s. We report the throughput results for all three workloads to indicate that parametrization trends are independent of workload.

As expected, parametrizing $\mathscr{H}$ with $(1,12)$ causes the least failure amplification, hence throughput is relatively unaffected by the failure at 25s. As n_1 grows larger, throughput grows steadily worse during the failure, because failure amplification becomes greater. We also point out that as the benchmark contains a larger percentage of reads in replex A, steady-state throughput increases (note the different Y-axis scales in Figure 13). This is because of the underlying LevelDB implementation of HyperDex. LevelDB is a simple key-value store with no secondary index support; reads on replex A are simple LevelDB gets, while reads to replex B become LevelDB scans. To achieve throughput as close to native gets as possible, we optimized point scans to act as gets to replex B, but the difference is still apparent in the throughput. Fortunately, this absolute difference in throughput does not affect the relative trends of parametrization.

The tradeoff from one parametrization to the next is throughput during failures in A. As an example, Figure 14 shows the throughput results when a machine in replex A is killed after 25s, with a 25:75 read workload. The performance of the parametrizations is effectively reversed. For example, even though $(1,12)$ performed best during a failure in B, it performs worst during a failure in A, in which failure amplification is 12x. Hence a user would select a parametrization based on which replex's failure performance is more valued.

5.4 Evaluating 3-Sharing

In the previous sections, all systems evaluated assumed 3-way replication. In particular, in Replex, if the number of indexes i specified by a table is less than 3, then Replex can build $3-i$ hybrid replexes for free, by which we mean those resources must be used anyway to achieve 3-way replication.

# Hybrids	Recovery Time (s)	Recovery Throughput (op/s)
0	14.7 ± 0.58	$5{,}831 \pm 678$
1	13.0 ± 0	$14{,}569 \pm 6{,}087$

Table 6: Recovery statistics for Replex systems with 3 replexes and different numbers of hybrid replexes. One machine is failed after 25 seconds. Results reported as average time $\pm$ standard deviation of 3 runs.

When $i \geq 3$, resource consumption from additional hybrid replexes becomes more interesting. No longer is a hybrid replex inserted to achieve a replication threshold; rather, a hybrid replex is inserted to increase recovery throughput, at the expense of an additional storage replica. Consider $i = 3$ and suppose a user only wishes to add a single hybrid replex, because of resource constraints. One way to maximize the utility of this hybrid replex is through 3-sharing, as described in Section 3.3. Of course, depending on the importance of the three original indexes, 2-sharing is also an option, but this is already explored in the previous sections. For sake of evaluation, we consider 3-sharing in this section.

The system under evaluation has 3 replexes, A,B,C, and 1 hybrid replex that is 3-shared across the original replexes. The hybrid replex is parametrized by $n_1 = 3$, $n_2 = 2$, $n_3 = 2$. Again, we load 1 million 1KB entries and fail a node at 25s. Four seperate client machines run a read benchmark spread equally across the indexes. Figure 15 shows the throughput results, compared to a Replex system without a hybrid replex.

Again, if there is no hybrid index, then recovery throughput suffers because of failure amplification. As long as a single hybrid index is added, the recovery throughput is more than doubled, with little change to recovery time. This experiment shows in the power of hy-

brid replexes in tables with more indexes: as the number of indexes grows, the fractional cost of adding a hybrid replex decreases, but the hybrid replex can still provide enormous gains during recovery.

6 Related Work

6.1 Erasure Coding

Erasure coding is a field of information theory which examines the tradeoffs of transforming a short message to a longer message in order to tolerate a partial erasure (loss) of the message. LDPC [25], LT [19], Online [18], Raptor [22], Reed-Solomon [20], Parity [8] and Tornado [17] are examples of well-known erasure codes which are used today. Hybrid replexes also explore the tradeoff between adding storage and network overheads and recovery performance. Recently, specific failure models have been applied to erasure coding to produce even more compact erasure codes [13]. Similarly, hybrid replex construction allows fine tuning given a workload and failure model.

6.2 Multi-Index Datastores

Several multi-index datastores have emerged as a response to the limitations of the NOSQL model. These datastores can be broadly divided into two categories: those which must contact every partition to query by secondary index, and those which support true, global secondary indexes. Cassandra [1], CouchDB [3], Hypertable [14], MongoDB [9], Riak [15] and SimpleDB [7] are examples of of the former approach. While these NOSQL stores are easy to scale since they only partition by a single "sharding" key, querying by secondary index can be particularly expensive if there is a large number of partitions. Some of these systems alleviate this overhead through the use of caching, but at the expense of consistency and overhead of maintaining the cache.

Unlike the previous NOSQL stores, Hyperdex [12] builds a global secondary index for each index, enabling efficent query of secondary indexes. However, each index is also replicated to maintain fault tolerance, which comes with a significant storage overhead. As we saw in Section 5, this leads to slower inserts and significant rebuild times on failure.

6.3 Relational (SQL) Databases

Traditional relational databases build multiple indexes and auxillary data structures, which are difficult to partition and scale. Sharded MySQL clusters [21, 10] are an example of an attempt to scale a relational database. While it supports fully relational queries, it is also plagued by performance and consistency issues [26, 10]. For example, a query which involves a secondary index must contact each shard, just as with a multi-index datastore.

Yesquel[2] provides the features of SQL with the scalability of a NOSQL system. Like Hyperdex, however, Yesquel separately replicates every index.

6.4 Other Data stores

Corfu [4], Tango [5], and Hyder [6] are examples of data stores which use state machine replication on top of a distributed shared log. While writes may be written to different partitions, queries are made to in-memory state, which allows efficient strongly consistent queries on multiple indexes without contacting any partitions. However, such an approach is limited to state which can fit in the memory of a single node. When state cannot fit in memory, it must be partitioned, resulting in a query which must contact each partition.

7 Conclusion

Programmers need to be able to query data by more than just a single key. For many NoSQL systems, supporting multiple indexes is more of an afterthought: a reaction to programmer frustration with the weakness of the NoSQL model. As a result, these systems pay unnecessary penalties in order to support querying by other indexes.

Replex reconsiders multi-index data stores from the bottom-up, showing that implementing secondary indexes can be inexpensive if treated as a first-order concern. Central to achieving negligible overhead is a novel replication scheme which considers fault-tolerance, availability, and indexing simultaneously. We have described this scheme and its parameters and have shown through our experimental results that we outperform HyperDex and Cassandra, state-of-the-art NoSQL systems, by as much as 10×. We have also carefully considered several failure scenarios that show Replex achieves considerable improvement on the rebuild time during failure, and consequently availability of the system. In short, we have demonstrated not only that a multi-index, scalable, high-availability NoSQL datastore is possible, it is the better choice.

References

[1] *Cassandra*. `http://cassandra.apache.org/`.

[2] Marcos K Aguilera, Joshua B Leners, Ramakrishna Kotla, and Michael Walfish. Yesquel: scalable SQL storage for Web applications. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, 2015.

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[4] Mahesh Balakrishnan, Dahlia Malkhi, John D Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4), 2013.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[6] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-a transactional record manager for shared flash. In *Proceedings of the Conference on Innovative Datasystems Research*, 2011.

[7] Andre Calil and Ronaldo dos Santos Mello. SimpleSQL: a relational layer for SimpleDB. In *Proceedings of Advances in Databases and Information Systems*. Springer, 2012.

[8] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 1994.

[9] Kristina Chodorow. *MongoDB: the definitive guide*. O'Reilly Media, Inc., 2013.

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3), 2013.

[12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4), 2012.

[13] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[14] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. *Department of Computer Science, Purdue University*, 2006.

[15] Rusty Klophaus. Riak core: building distributed applications without shared state. In *Proceedings of ACM SIGPLAN Commercial Users of Functional Programming*, 2010.

[16] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, 2009.

[17] Michael Luby. Tornado codes: Practical erasure codes based on random irregular graphs. In *Randomization and Approximation Techniques in Computer Science*. Springer, 1998.

[18] Petar Maymounkov. Online codes. Technical report, New York University, 2002.

[19] Thanh Dang Nguyen, L-L Yang, and Lajos Hanzo. Systematic luby transform codes and their soft decoding. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2007.

[20] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2), 1960.

[21] Mikael Ronstrom and Lars Thalmann. MySQL cluster architecture overview. *MySQL Technical White Paper*, 2004.

[22] Amin Shokrollahi. Raptor codes. *IEEE Transaction on Information Theory*, 52(6), 2006.

[23] Jeff Terrace and Michael J Freedman. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *Proceedings of the USENIX Annual Technical Conference*, 2009.

[24] Alexander Thomson and Daniel J Abadi. CalvinFS: consistent WAN replication and scalable metadata management for distributed file systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[25] Jeremy Thorpe. Low-density parity-check (LDPC) codes constructed from protographs. *IPN progress report*, 42(154), 2003.

[26] Bogdan George Tudorica and Cristian Bucur. A comparison between several NoSQL databases with comments and notes. In *Proceedings of Roedunet International Conference*. IEEE, 2011.

[27] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2004.

Kinetic Modeling of Data Eviction in Cache

Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo
Peking University

Chen Ding
University of Rochester

Zhenlin Wang
Michigan Technological University

Abstract

The reuse distance (LRU stack distance) is an essential metric for performance prediction and optimization of storage and CPU cache. Over the last four decades, there have been steady improvements in the algorithmic efficiency of reuse distance measurement. This progress is accelerating in recent years both in theory and practical implementation.

In this paper, we present a kinetic model of LRU cache memory, based on the average eviction time (AET) of the cached data. The AET model enables fast measurement and low-cost sampling. It can produce the miss ratio curve (MRC) in linear time with extremely low space costs. On both CPU and storage benchmarks, AET reduces the time and space costs compare to former techniques. Furthermore, AET is a composable model that can characterize shared cache behavior through modeling individual programs.

1 Introduction

A memory system is a multi-level structure where the upper level of memory often plays the role of cache for the lower level of storage. This design is motivated by a simple fact of *program locality*: in any time period, only a small fraction of data in a program will be frequently used. This behavior used to be modeled by the working set locality theory [1] where data locality is characterized by working set size (WSS) [2, 3]. Locality characterization techniques have been developed for decades. They are widely used for management and optimization at different levels of memory hierarchy.

Much progress has been made to model locality through reuse distance analyses and the result miss ratio curves (MRCs), as shown in Figure 1. From the reference trace of a program, accurate MRC can be calculated by measuring reuse distance (LRU stack distance as defined by Mattson et al. [4]). Reuse distance is the number of distinct data accesses between two consecutive accesses to the same location. Precise reuse distance tracking requires $O(N \log M)$ time and $O(M)$ space for a trace of N accesses to M distinct elements [5].

For CPU workloads, the recent footprint theory [6], StatStack [7] and time-to-locality conversion [8, 9] use *reuse time* instead of reuse distance to model the workloads. Reuse time is the time between an access and its next reuse. The footprint approach reduces the run-time overhead of MRC measurement to $O(N)$.[1] However, the space overhead of the footprint algorithm is still $O(M)$.

As for storage workloads, their sizes are usually much larger than CPU workloads and their life span may last for weeks or more. Therefore, techniques like the footprint analysis may require too much space. *Counter Stacks* [11] and *SHARDS* [12] are recent breakthroughs to reduce space cost in asymptotic complexity [11] and in practice [12]. Counter Stacks uses probabilistic counters and for the first time can measure reuse distances in sublinear space with a guaranteed accuracy [13]. SHARDS

[1] The working-set theory has a similar effect and same time and space complexity [10, 3]. See Sec. 2.8 of [6] for a comparison.

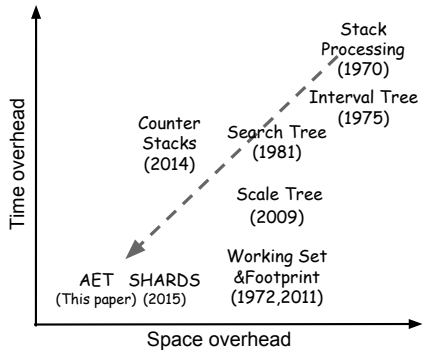


Figure 1: Time and space cost of MRC profiling algorithms.

uses a splay tree to track the reuse distances of sampled data. The time and space consumption is reduced to an extremely low level. However, Counter Stacks and SHARDS cannot characterize shared cache behavior through modeling individual programs.

This paper describes a new kinetic model for MRC construction of LRU caches based on average eviction time (AET). AET runs in linear time asymptotically and uses sampling to minimize the space overhead. In evaluation, AET has the lowest level space and run-time overhead compared to past techniques, for both CPU workloads and storage workloads, while maintaining high MRC accuracy. Although SHARDS is comparable to AET in time and space overhead, AET is a composable metric, i.e. the MRC of a multi-programmed workload in shared cache can be computed directly from the AET of its member programs.

2 AET Model

This section describes the kinetic model. Section 2.1 uses an example to introduce the basic concepts especially the eviction time. Section 2.2 formulates and computes the average eviction time (AET) by solving the distance integration equation. Section 2.3 discusses the correctness of the model. Section 2.4 models the shared cache and solves the eviction-time equalization equation.

2.1 LRU Stack and Eviction Time

LRU cache can be logically viewed as a stack [4]. Data blocks are ranked by their recent access time from most recent to least recent. Every access brings the accessed data to the top of the stack. The bottom of the stack stores the least recently used data and is evicted on a miss (when the cache is full).

When a data block is loaded into cache on a miss, it may be reused for several times (hits) before it is evicted. The *eviction time* is the time between the last access and the eviction. It is the duration that the block moves from the top of the stack to the bottom for the *last* time. At an eviction at time t, looking backwards to the most recent time u when the evicted block was referenced, the time interval $t-u$ is the eviction time. Notice that u could also be the time the data block was brought in (a miss). In general, the eviction time is the last segment of the residence time of the data block.

For example, block d in the cache in Figure 2 is loaded at time 3, last accessed at time 5, and evicted at time 10. The eviction time is 5, shown by the shaded area.[2]

To model the eviction time, we need to model the progression that leads to the eviction. We define the *arrival time* T_m as the time it takes for a block to reach stack position m (from its last access). For size c cache, the arrival time is a (subscripted) function $T_m, m = 0, \ldots, c-1$. Naturally, $T_0 = 0$ and the eviction time is T_c, which is the time the data block leaves position $c-1$. To illustrate, Table 1 shows the arrival time T_m of d for size 4 cache. As m increments from 0 to 3, T_m increases from 0 to 5.

[2]Eviction time is part of the *residence time*, which can be estimated using queuing theory (as "response time", Chapter 9 [14]).

time	ref		rt	hit
0	a	a	∞	0
1	b	b → a	∞	0
2	c	c → b → a	∞	0
3	d	d → c → b → a	∞	0
4	a	a → d → c → b	4	1
5	d	d → a → c → b	2	1
6	a	a → d → c → b	2	1
7	b	b → a → d → c	6	1
8	a	a → b → d → c	2	1
9	c	c → a → b → d	7	1
10	e	e → c → a → b	∞	0
11	d	d → e → c → a	6	0

Figure 2: Example 4-block cache, viewed as a stack, showing logical time, data referenced each time (ref), reuse time (rt) of each access and whether the access is a cache hit. The shaded area is the eviction time of d.

The movement of block d depends on how other data are accessed. At each access in the eviction process (shaded area in Figure 2), d either stays at its current position or steps down one position. The *condition of movement* is simple: d moves down from a position m if and only if the access is a miss, or if the stack position of the accessed data m' is greater than m, that is, lower in the stack. We define T_0 to be 0. Obviously, T_1 is al-

Table 1: The kinetic model illustrated by d's eviction in the shaded area in Figure 2. The arrival time T_m (third row) depends on the movement condition: whether the reuse time (last row) is greater than T_m. The eviction time is $T_4 = 5$.

Logical time	5	6	7	8	9	10
Position m	0	1	2	2	3	evicted
Arrival time T_m	0	1	2	2	4	5
Current reuse time	2	2	6	2	7	∞

ways 1, since the access to any other block must bring it to stack position 0 and dislodge d, as it happens at time 6 for d.

The condition of movement can be simplified, because we do not need the exact location of the accessed data. It suffices to know the relative location. For a simpler test, we use the reuse time rather than the stack location. When block z is accessed, and d is at stack position m, d moves down if and only if the (backward) reuse time of z is greater than d's arrival time T_m.

The relation between the eviction time and the reuse time is illustrated by our example. The last row of Table 1 shows the reuse time of each access during d's movement. Block d moves its position (shown in the second row) whenever the reuse time (the last row) is greater than the arrival time (the third row).

We next model the average eviction time for all data in cache. The arrival time T_m will be defined similarly as the average for all data.

2.2 Average Eviction Time (AET)

$AET(c)$ is the Average Eviction Time for all data evictions in a fully associative LRU cache of size c. T_m is the average arrival time for a data block to reach position m (in its eviction process). Obviously, $T_0 = 0$ and $AET(c) = T_c$. The movement condition is no longer individual but now collective and depends on the reuse times of all data.

Let n be the total number of references and $rt(t)$ be the number of references with reuse time t. $f(t)$ is the proportion of reuses with reuse time t, defined as follows.

$$f(t) = \frac{rt(t)}{n} \tag{1}$$

For an access, $P(t)$ is the probability that its reuse time is greater than t:

$$P(t) = \sum_{t+1}^{\infty} f(t) \tag{2}$$

The movement condition is now a probability. It is actually $P(t)$. This can be interpreted as follows: in a unit time, a data block moves by $P(t)$ position. To use a familiar concept, we call it the *travel speed*. At position m, the average arrival time is T_m, and the travel speed $v(T_m)$ is the probability in logical time:

$$v(T_m) = P(T_m) \tag{3}$$

For a given block at each stack position, the moving speed is easy to define: either moving one position at the next access or stay in place (no movement). This travel speed may slow down and then speed up. On average for all evictions, however, the velocity is monotone and non-increasing. By definition, $P(T_m)$ is monotone and non-increasing with T_m. It follows from Eq. 3 that the travel speed at position m is monotone and non-increasing with m.

We now construct an equation to solve for T_m and then $AET(c)$. The equation connects three metrics: velocity $v(T_m)$, average arrival time T_m, and cache size c. This connection is shown pictorially in Figure 3.

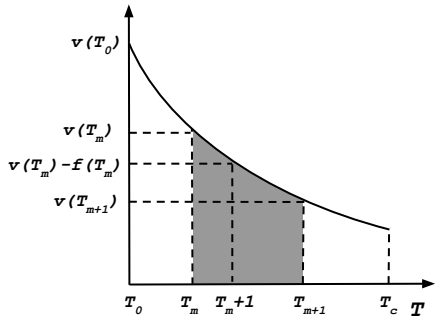


Figure 3: As the average arrival time (T_m) increases along the x-axis, the y-axis shows the travel speed $v(T_m)$ at each T_m. The integral of of v over T gives the movement distance, which is the area under the curve. The shaded area shows the increment of stack position (which is 1).

In Figure 3, the x-axis shows the average arrival time (T_m) as it increases. At each T_m, we use Eq. 3 to compute the travel speed $v(T_m)$, shown in the y-axis. The figure shows an example curve, which is monotonically non-increasing. The integral of v over T gives the movement distance, i.e. the stack position it travels to. It is the area under the curve. The shaded area shows the increment of the stack position (which is 1).

The three metrics are discrete functions. The subtle but critical problem is the difference in their discrete units. When we measure the cache size and the data movement in cache, a single step is a stack position. When we measure the reuse time, a single step is an access. We may call the former the spatial unit and the latter the temporal unit. The two units are not the same. Figure 3 shows that from the same base T_m, the temporal increment $T_m + 1$ is less than or equal to the spatial increment T_{m+1}.

We use the temporal-unit function of reuse time to derive the spatial-unit function of AET. Let's consider how the speed changes as a data block travels. From the monotonicity mentioned earlier, the change must be a deceleration. Based on the velocity formula (Eq. 3), the following gives the exact deceleration from T_m to $T_m + \Delta T$.

$$v(T_m + \Delta T) = v(T_m) - \sum_{t=T_m}^{T_m+\Delta T-1} f(t), \tag{4}$$

where ΔT stands for the time increase over T_m. The unit is temporal, so the minimal ΔT is one, i.e. one access.

Now we are ready to formulate the first kinetic equation, Distance Integration (DI). It combines the temporal and spatial increments to compute the complete movements. First, let's consider the spatial increment. From T_m to T_{m+1}, the data travels one stack position (the shaded area in Figure 3). Second, we add the temporal increment as follows. For each spatial increment (m), we compute the deceleration by integrating in the temporal unit (dx), given in Eq. 4. Finally, we sum over the spatial increment from 0 to cache size c. The result is the total distance traveled, e.g. the area below the example curve in Figure 3, which is the cache size c when the arrival time reaches T_c.

$$\sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (v(T_m) - \sum_{t=T_m}^{x-1} f(t))\,\mathrm{d}x = c \tag{5}$$

DI is an implicit equation. Its solution, as it turns out, is $AET(c)$. Consider the speed at each time step x from 0 to $AET(c)$, and the time it takes at each step, we have:

$$\int_0^{AET(c)} P(x)\,\mathrm{d}x = c \tag{6}$$

This equation is in fact the same as Eq. 5. The equivalence is proved as follows.

$$\begin{aligned}
&\sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (v(T_m) - \sum_{t=T_m}^{x-1} f(t))\,\mathrm{d}x \\
&= \sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (P(T_m) - \sum_{t=T_m}^{x-1} f(t))\,\mathrm{d}x \\
&= \sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} (P(T_m) - (P(T_m) - P(x)))\,\mathrm{d}x \\
&= \sum_{m=0}^{c-1} \int_{T_m}^{T_{m+1}} P(x)\,\mathrm{d}x \\
&= \int_{T_0}^{T_1} P(x)\,\mathrm{d}x + \int_{T_1}^{T_2} P(x)\,\mathrm{d}x \\
&\quad \ldots + \int_{T_{c-1}}^{T_c} P(x)\,\mathrm{d}x \\
&= \int_0^{AET(c)} P(x)\,\mathrm{d}x
\end{aligned}$$

From AET to MRC Eq. 6 shows that AET calculation takes linear time. The only information it needs is the reuse time histogram (RTH), which gives $P(x)$, and can be measured in linear time. The miss ratio $mr(c)$ at cache size c is the probability that a reuse time is greater than the average eviction time $AET(c)$:

$$mr(c) = P(AET(c)) \tag{7}$$

During the integration of Eq. 6 from 0 to maximal reuse time, the miss ratios of all cache sizes can be computed in linear time at once.

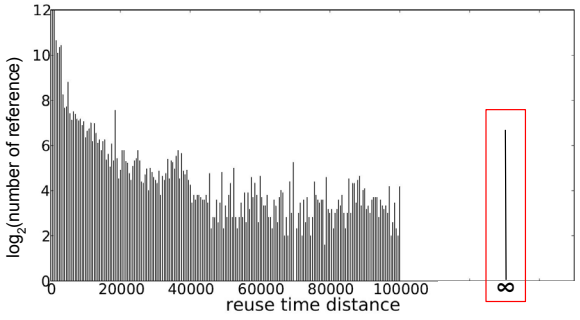


Figure 4: RTH and cold miss example

Impact of Cold Misses In a program execution, the first access to any data block should be a cold miss. Because every cold miss will insert a new data block at the head of the LRU priority list, it will push down all the data in the list by one position. In the kinetic equation, no matter where the data is, the cold misses always contribute a fixed share of probability that moves the data. Therefore, in AET model, we define the reuse time of every cold miss to be infinite, and we count the number of cold misses in the ∞ bin of the reuse time histogram (RTH), as in the example shown in Figure 4.

2.3 Correctness

The conversion from AET to miss ratio is not always correct. The correct miss ratio for cache size c is the proportion of reuse distances $d > c$.

The inverse of the AET function is in fact an estimation of reuse distance. For a reuse time t, the reuse distance d is the distance the data block traveled down the cache stack, so $t = AET(d)$ and:

$$d = AET^{-1}(t) \tag{8}$$

AET conversion is equivalent to first estimating the reuse distance and then using the estimated reuse distance:

$$\begin{aligned}
mr(c) &= \frac{\sum_{x>c} rd(x)}{n} \\
&= \frac{\sum_{t>AET(c)} rd(AET^{-1}(t))}{n} \\
&= \frac{\sum_{t>AET(c)} rt(t)}{n} \\
&= P(AET(c)),
\end{aligned}$$

where $rd(x)$ is the number of references with reuse distance x. Therefore, AET is correct if its estimation of reuse distance is correct. Hence:

Correctness Condition. *The AET-based conversions are accurate if the number of reuse times $rt(t)$ of time t is the same as the number of reuse distances $rd(AET^{-1}(t))$ of distance $AET^{-1}(t)$, for all $t > 0$.*

When the two are equal, using the AET conversion is the same as using reuse distance for all cache size $c \geq 0$. The condition is a reiteration of Eq. 8 but shows the connection mathematically as a function composition.

2.4 AET in Shared Cache

When sharing the cache, a set of co-run programs interact. We want a composable model to derive the composite effect from individual solo-run locality. Ding et al. [15] define the composability as follows: a locality metric is composable if the metric of a co-run can be computed from the metric of solo-runs. AET is composable: given the solo-run AETs of individual programs, we can derive the co-run AETs in the shared cache. There are $n+1$ co-run AETs for n co-run programs: one for each program and one for the group. We derive them by solving another AET equation. Equation solving has two basic questions: does a solution exist, and if so, is the solution unique?

Cache sharing means that *all co-run programs have the same average eviction time (AET).* For any data block in the shared cache, once it is no longer accessed, its eviction time is the same regardless which program the data block belongs to. Hence we have the equation of eviction-time equalization: *when n programs share the cache of size c, all $n+1$ co-run AETs, $AET_i(c)$ for each program i and $AET(c)$ for the group, are the same*:

$$AET_1(c) = AET_2(c) = \cdots = AET_n(c) = AET(c) \quad (9)$$

We now show that this equation has one and only one solution.

To explain the derivation we start with the symmetrical case, where n co-run programs are identical. Let r_{solo} be the access rate, $rt_{solo}(t)$ be the reuse-time histogram, $P_{solo}(t)$ be the probability function, defined as in Section 2.2 for each program. The aggregate access rate is naturally $r_{co} = n\, r_{solo}$. We define the *co-run logical clock*. The co-run clock runs n times faster, with one out of every n ticks for each program. For each program, the co-run reuse time $rt_{co}(nt) = rt_{solo}(t)$, or equivalently $rt_{co}(t) = rt_{solo}(t/n)$. Because of the time change, the probability function of each program becomes $P_{co}(t) = P_{solo}(t/n)/n$. The aggregate probability is the sum of the group, $P(t) = \sum_{i=1}^{n} P_{co}(t) = nP_{co}(t)$.

$$P(t) = \sum_{i=1}^{n} P_{co}(t) = \sum_{i=1}^{n} P_{solo}(t/n)/n = P_{solo}(t/n) \quad (10)$$

From $P(t)$, we use the distance-integration equation (Eq. 6) to derive the co-run AET:

$$\int_0^{AET(c)} P(x)\,\mathrm{d}x = c \quad (11)$$

The equation looks the same as Eq. 6, but $P(x)$ is the aggregate probability, x is the co-run time, and $AET(c)$ is average eviction time of the shared cache.

In the shared cache, any access by any program is a miss if and only if its reuse time is greater than $AET(c)$. The group miss ratio is therefore $mr(c) = P(AET(c))$, and the portion of this miss ratio contributed from each program is $mr_{co}(c) = P_{co}(AET(c))$. This contribution is the same from every program, so $mr_{co}(c) = mr(c)/n$. The solutions of the co-run AET and miss ratio for this symmetric case are unique.

Note that the co-run miss ratio $mr_{co}(c)$ is the ratio of the miss count of each program divided by the number of accesses of all programs. In other words, it is the miss ratio defined on the co-run clock. This definition enables us to add miss ratios of different programs directly. It can also be easily converted to the conventional miss ratio.

We now consider the general case. It differs from the previous, symmetric case in two ways: each program i may have a different access rate $r_{solo,i}$ and a different reuse time histogram and hence the probability function $P_{solo,i}(t)$. Let the total access rate be $r = \sum_{i=1}^{n} r_{solo,i}$. The aggregate $P(t)$ is:

$$P(t) = \sum_{i=1}^{n} P_{co,i}(t) = \sum_{i=1}^{n} P_{solo}(t\frac{r_{solo,i}}{r})\frac{r_{solo,i}}{r} \quad (12)$$

The shared-cache distance-integration equation (Eq. 11) can now compute $AET(c)$ for the general case. The group miss ratio is $mr(c) = P(AET(c))$, and the portion of the miss ratio contributed from program i is $mr_{co,i}(c) = P_{co,i}(AET(c))$. The contribution is now individualized and differs depending on the individual access rate $r_{solo,i}$ and reuse time histogram $rt_{solo,i}(t)$. Below is the co-run miss ratio of the group as the sum of the co-run miss ratio of each individual. These solutions are unique for each program group.

$$mr(c) = P(AET(c)) = \sum_{i=1}^{n} mr_{co,i}(c) = \sum_{i=1}^{n} P_{co,i}(AET(c)) \quad (13)$$

Composition Invariance The aggregated miss ratio can be computed using AET in two ways: directly using the group $P(t)$ or indirectly as the sum of individual miss ratios. Mathematically, the two results are the same, as shown by Eq. 13. We call this mathematical equivalence the *composition invariance*. A composable model has this invariance if the group miss ratio is the same whether it is composed from the individual (solo-run) locality or added together from the individual (co-run) miss ratio. Early composable models used reuse distance and footprint and had only one way to compute the group miss ratio [16, 17, 18, 19]. Recent models used footprint and the higher order theory of locality (HOTL) to obtain composition invariance [6, 20, 21]. The model by Brock et al. treated the shared cache as the partitioned cache, where each program is "imagined" to occupy a *natural partition* [21]. AET obtains composition invariance using eviction-time equalization. Unlike the "imagined" natural partition, eviction-time equalization is a real property of the shared cache.

3 Reuse Time Histogram (RTH) Sampling

For efficiency, AET-based MRC profiling can use sampled RTH instead of real RTH. Since it is only the probability distribution that it cares about, if the sampled RTH maintains the same distribution as the real RTH, the estimated AET will be accurate. By sampling a small fraction of references, the space overhead can be largely eliminated. This section presents efficient MRC analysis through AET sampling.

3.1 Sampling Techniques

In order to capture the distribution of the real RTH, all the references have to be sampled with equal probability. This seems to be an easy target, but it is not the case in real applications. Next, we list four sampling techniques and discuss their strength and weakness.

Address Sampling The address sampling requires monitoring a fixed subset of the address space. It is known as hold-and-sample and has been used in measuring reuse distance [22, 23, 24, 25] or reuse time [26]. During sampling phase, all the references to the subset will be recorded in sampled RTH. This technique is simple and easy to implement, and only a fixed hash table is required. However, in a real program, references are not evenly distributed on every data object. Large portion of accesses may focus on a small subset. In this case, the RTH collected from a small portion of working set may not reflect the real reference pattern. This will lead to imprecise estimation of AET.

Fixed Interval Sampling To avoid the bias of address sampling, the fixed interval sampling collects a subset of references instead of a subset of the address space. After every m references, it places the current accessed data into the monitoring set. At the next reference of the data, the reuse time is recorded into RTH, and the data is deleted from the monitoring set. By this design, the reuses are sampled by the same probability, which provides a better RTH approximation than address sampling. However, the accuracy of fixed interval sampling may be influenced by another problem. Since the sampling rate m is a fixed value, if the reference pattern of some data shows a different distribution at the chosen interval, the sampled RTH cannot reflect the actual distribution of this pattern.

Random Sampling The random sampling can overcame the problem we mentioned in fixed interval sampling and address sampling. Instead of using fixed sampling rate m, the distance between two adjacent monitoring points is a random value. In a real application, we can set the random value to a certain range to control the number of references sampled for RTH. We have tested the above three sampling techniques and found that the random sampling achieved the highest stability and accuracy. This form of random sampling for MRC analysis is pioneered by StatStack [7].

Reservoir Sampling The space used to store sampled data grows linearly. To bound the space cost, reservoir sampling technique [27] was used by Beyls and D'Hollander [26] for locality analysis. Let the number of entries in the monitoring set (reservoir) be k. When the i-th sampled data arrives, reservoir sampling keeps the new data (tagged as "unsampled") in set with probability $min(1,\ k/i)$ and randomly discards an old data block when the set is full. Every time a monitored data block is reused, its reuse time will be recorded. This data block will be tagged as "sampled" and all of its following reuses will not be recorded. This design ensures even sampling and avoids the access distribution problem we have in address sampling. When the sampling is over, the RTH is updated based on the "sampled" data entries remaining in set. The "unsampled" entries are those data objects with no reuse after being inserted. They are cold misses which we will discuss in Section 3.3. Reservoir sampling reduces the space complexity of RTH sampling from $O(M)$ to $O(1)$.

3.2 Phase Sampling

For programs that have an unstable reference pattern, we evenly divide its execution into phases. For each phase, we use random sampling to construct the RTH and MRC

for this phase. Then we construct the MRC of the entire program. The miss ratio at any cache size is average miss ratio of all MRCs at this size. We call this technique phase sampling.

Phase sampling is used by StatStack [7]. To adapt it for AET sampling, there is one important design change. Not every sampled data in the monitoring set will see its reuse in the same phase. Before entering the next phase, the monitoring set will not be cleaned, the next phase still keeps track of these data until they are reused and then deleted from the monitoring set. We use the backward reuse time, so the inter-phase reuse time is added to the RTH of the current phase. In contrast, StatStack uses the forward reuse time. A second and more significant difference with StatStack is the handling of cold misses.

3.3 Cold Miss Handling

As we mentioned in Section 2.2, the ∞ bin of RTH counts the number of cold misses. Therefore, we should set the infinite reuse-time bin of the sampled RTH to the number of cold misses in all sampled references. However, in random sampling, we cannot tell if a sampled access is the first reference to an address. As we know, in a trace of finite length, any referenced address has its first access and last access. It means the number of cold misses is equal to the number of the references that have no reuse (last access). Because the chances to meet these two kinds of access are equal, we use the number of references with no reuse in all sampled references to revise the number of cold misses in the sampled RTH. In random sampling, they are the data objects that are still in the monitoring set after sampling is complete. In reservoir sampling, they are the data objects that are tagged "unsampled".

4 Evaluation

In this section, we evaluate the AET model by comparing it with four recent techniques: Counter Stacks [11], SHARDS [12], StatStack [7] and adaptive bursty footprint (ABF) sampling [20]. The first two are for storage workloads, while the last two are for CPU workloads.

We use a Dell PowerEdge R720 with ten-core 2.50GHz Intel Xeon E5-2670 v2 processors and 256 GB of RAM. Benchmark traces are read from RAMDisk to avoid the IO bandwidth delay. We have implemented these techniques in C++. To save memory and make a fair comparison, we record the reuse time histogram (AET, StatStack, ABF sampling) and reuse distance histogram (Counter Stacks, SHARDS) using the compressed representation by Xiang et al. [19] Each histogram is an array which is binned in logarithmic ranges. Each (large enough) power-of-two range is divided into (up to) 256 equal-size increments. This representation requires less than 100KB for all our workloads.

4.1 AET vs Counter Stacks

Counter Stacks is a recent algorithmic breakthrough by Wires et al. to finally solve the open problem of reducing the asymptotic space complexity of MRC analysis to below M, the size of data [11]. It uses probabilistic counters to estimate the reuse distances. While other reuse distance measurement techniques consume linear space overhead, the HyperLogLog counter [28] used by Counter Stacks only requires extremely small space while maintaining an acceptable accuracy. Every d references and every s seconds, Counter Stacks starts a new counter to record the number of distinct data accessed from the current time. During the execution, the number of active counters keeps growing. Counter Stacks periodically writes the results of active counters to the disk. The data in the disk is used to compute the reuse distance distribution and construct MRC. To reduce the number of live counters, Counter Stacks uses a pruning strategy to delete a younger counter whenever its value is as least $(1-\delta)$ times the older counter's value. By controlling δ, Counter Stacks can balance between accuracy and number of counters.

We compare AET model with Counter Stacks using the same storage traces released by Microsoft Research Cambridge (MSR) [29], as used by Counter Stacks. The traces are configured with only read requests of 4KB cache blocks. We test Counter Stacks under two different fidelities. The experimental parameters follow those used in [11], with high fidelity ($d = 1M$, $s = 60$, $\delta = 0.02$) and low fidelity ($d = 1M$, $s = 3600$, $\delta = 0.1$). For AET, we use random sampling at the rate 10^{-4}, and reservoir sampling where the number of entries in the hash table (32-bit address) is limited to 16K. Figure 5 shows the MRCs profiled by AET random sampling and high fidelity Counter Stacks (CS-high) as well as the real MRCs calculated using precise reuse distances. As we can observe, AET sampling and CS-high both approximate the real MRCs well. As for CS-low and AET reservoir sampling, we only list their absolute prediction error in Table 2 for comparison.

Table 2 shows two types of averages, arithmetic and weighted. The ones marked with a '*' are weighted by the working set size, which is the length of MRC. The weighted average prediction errors of AET random sampling (RAN, 0.96%) and AET reservoir sampling (RES, 1.12%) are in between of high fidelity Counter Stacks (0.77%) and low fidelity Counter Stacks (1.26%) but they show much higher throughput (arithmetic average) and much lower space overhead (weighted average) than both methods of Counter Stacks.

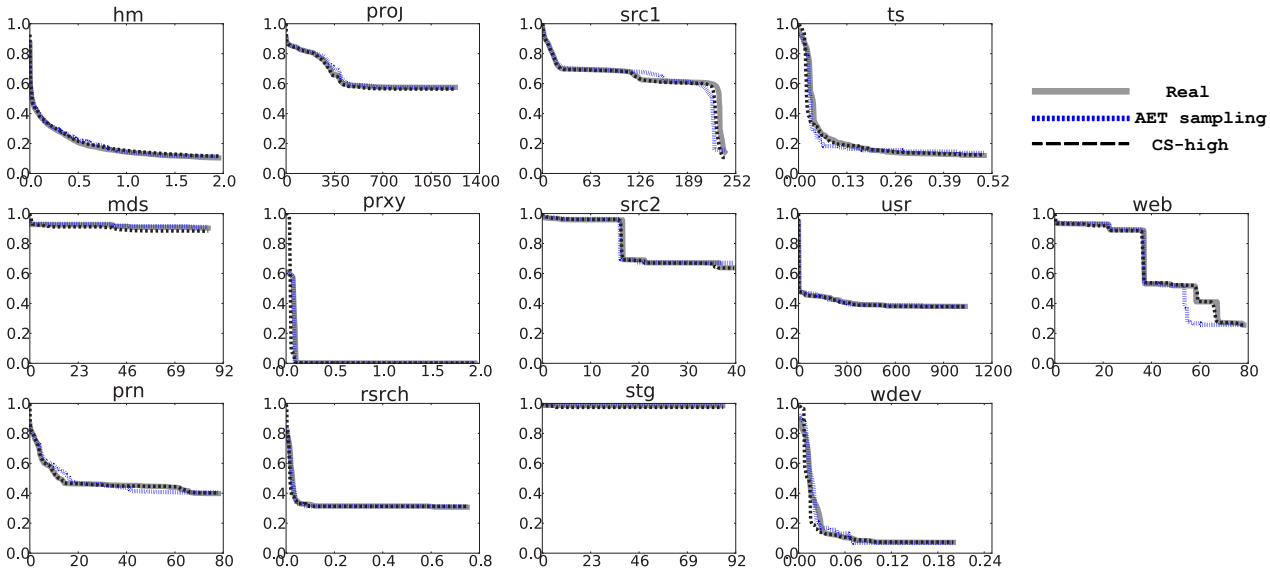


Figure 5: The predicted miss ratio (y-axis) over cache size (GB, x-axis) by AET sampling and Counter Stacks

AET uses reuse time histograms while Counter Stacks uses reuse distance histograms. No matter what compression technique is used for the histogram, the size of both histogram structures should be comparable. Consequently, the key difference in space between the two techniques is the hash table used by the AET algorithm and the Hyperloglog counters used by Counter Stacks. In AET random sampling, the number of hash table entries is the number of data blocks being monitored at this time. The theoretical upper bound is the working set size times the sampling rate. In AET reservoir sampling, the space is constant, i.e. a hash table of a fixed size. In Counter Stacks, the space used by probabilistic counters grows when more counters are used. Therefore, the space overhead of Counter Stacks is not constant. In Table 2, we also list the memory consumed by the hash table and Hyperloglog counters for the MSR traces. The results show that the actual memory usage of AET random sampling is much lower than Counter Stacks. In fact, the total space consumption (not including the histogram array) of all 13 traces by AET random sampling is 2.2MB, while low and high fidelity Counter Stacks require 11MB and 56MB for Hyperloglog counters, respectively. In AET reservoir sampling, the space overhead is fixed at 384KB for each trace. Although the overall space consumption (5MB) is larger than random sampling, its weighted average space overhead is less than random sampling. Reservoir sampling reduces the space cost of random sampling only in *proj* and *usr*. They are the traces with the largest working set sizes. The remaining traces have smaller working sets. For these traces, reservoir sampling incurs a higher error even when it uses more space than random sampling. As we mentioned in Section 3.1, reservoir sampling only uses the remaining entries in hash table to update RTH and does not delete the data entry after the reuse is sampled (in order to measure the cold miss ratio). The actual number of reuses in RTH of reservoir sampling is less than random sampling under the same sampling rate.

It takes Counter Stacks $O(\log M)$ time to update the counters at each reference and $O(N \log M)$ for the entire trace. AET is linear time in $O(N)$. Table 2 shows that in our implementation, the throughput of AET random sampling is 37 and 11 times of the throughput of high and low fidelity Counter Stacks, respectively. AET reservoir sampling shows a similar throughput as AET random sampling does.

The correctness of AET-based MRC is based on the assumption of stable distribution reuse distances. This brings inaccuracies to those data that violate the assumption. As we can observe in Figure 5 the AET-based MRC of *web* mispredicts the knee at around 50GB, but Counter Stacks perfectly models every details of the curve, since it makes no assumption about the data at all. Now we can clarify the trade-off between the two techniques: AET makes a statistical assumption, offering good accuracy in most cases in $O(N)$ time. Counter Stacks makes no statistical assumption, delivering good accuracy in all cases in $O(N \log M)$ time.

4.2 AET vs SHARDS

SHARDS (Spatially Hashed Approximate Reuse Distance Sampling) is recently developed by Waldspurber et al. [12]. It uses hash-based spatial sampling and a splay tree to track the reuse distances of the sampled data. It

Table 2: The comparison between Counter Stacks (CS) and AET

	WSS (GB)	Prediction Error (%)				Memory (KB)				Throughput (*M*reqs/sec)			
		AET		CS		AET		CS		AET		CS	
		RES	RAN	high	low	RES	RAN	high	low	RES	RAN	high	low
proj	1238.9	0.76	0.74	0.93	1.04	384	584	8384	1376	31.01	26.10	1.32	3.94
usr	1035.1	0.79	0.37	0.24	0.31	384	501	7744	1376	30.22	30.67	1.36	3.87
src1	312.7	3.09	2.90	1.54	4.78	384	176	5408	1088	30.17	44.88	1.86	4.88
mds	86.9	0.85	0.70	1.81	1.82	384	114	2848	832	79.82	77.08	3.16	6.17
stg	85.7	0.09	1.01	1.11	1.11	384	114	4256	928	78.90	51.99	2.23	6.30
web	78.3	3.81	3.65	1.00	2.92	384	111	6464	1120	56.00	70.67	1.50	5.60
prn	77.5	2.28	2.08	0.31	0.57	384	110	4960	960	60.81	71.17	1.28	5.79
src2	39.9	1.09	1.02	0.57	2.19	384	94	4704	960	84.49	71.44	2.48	6.66
hm	2.0	0.90	0.77	1.01	1.31	384	79	3680	608	65.74	67.62	0.33	6.87
prxy	2.0	0.20	0.04	1.62	1.69	384	79	2112	576	31.43	76.77	3.40	7.23
rsrch	0.7	2.90	0.92	0.30	2.84	384	78	2720	416	82.55	82.55	1.22	7.26
ts	0.5	1.51	2.04	0.41	0.78	384	78	1920	640	88.02	74.12	1.08	5.80
wdev	0.2	2.62	1.21	0.20	0.11	384	78	864	352	86.81	86.81	1.28	5.75
avg*	-	1.12*	0.96*	0.77*	1.26*	384*	452*	7363*	1292*	61.99	63.99	1.73	5.86
sum	2960	-	-	-	-	4992	2196	56066	11232	-	-	-	-

limits the space overhead to a constant by adaptively lowering the sampling rate. SHARDS outperforms Counter Stacks in both memory consumption and throughput for the merged "master" MSR trace (created by Wires et al [11]), which is a 2.4 billion-access trace combining all 13 MSR traces by ranking the time stamps of all accesses. Following them, we use the master trace for evaluation. For fairness comparison, we let AET and SHARDS both use $8K$ buckets hash table (64-bit address) for sampling. The pointers and variables in our implementation are all 64-bit sizes. Figure 6 shows the MRC profiled by AET random sampling with sampling rate $1 * 10^{-6}$. The Mean Absolute Error (MAE) is 0.01. SHARDS gives a lower MAE of 0.006 with $8K$ samples. We check the peak resident memory usage at run time, AET random sampling consumes 1.7MB memory while SHARDS consumes 2.3MB memory. The throughput of AET and SHARDS are 79.0M blocks/sec and 81.4M blocks/sec respectively. For the same trace, Counter Stacks is most accurate, with an MAE of 0.003. However, it consumes 80MB memory, and the throughput is relatively low, 2.3M blocks/sec [11]. AET reservoir sampling ($8K$) uses 1.4MB resident memory with 66.6M blocks/sec and an MAE of 0.01, same as AET random.

SHARDS and AET sampling have same time and space complexity, and their run time and memory usage are close in our test. However, the applicability of SHARDS is not limited to miss ratio prediction of LRU caches. Waldspurber et al. [12] showed that the hash-based spatial sampling technique of SHARDS can be used to perform efficient scaled-down simulations for non-LRU caching algorithms such as ARC [30]. Since

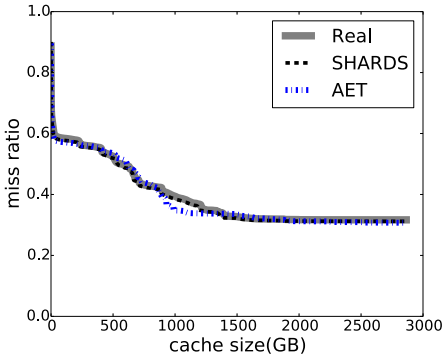


Figure 6: MRCs predicted by AET sampling and SHARDS for the master trace

AET sampling is tied with LRU caches, cache simulations for non-LRU algorithms cannot be done by the current AET model. The strength of AET model is composability, which can be used to model shared cache as we will show in Section 4.6. But this is not a property of current SHARDS and Counter Stacks.

4.3 AET vs StatStack

StatStack is one of the most efficient and accurate methods to approximate MRC for CPU workloads. It samples cache blocks and measures their reuse time using hardware and operating system support such as performance counters and watchpoints [31, 32]. From the reuse time distribution, StatStack estimates the capacity miss ratio and predicts the real miss ratio by adding the estimated

cold miss ratio. We use the SPEC CPU2006 benchmark suite [33] to compare AET and StatStack. For each benchmark, we intercept 1 billion references from their execution using the instrumentation tool Pin [34].

In Figure 7, we show the cumulative distribution function (CDF) of the absolute error for both techniques as well as AET random sampling technique under two sampling rate of 10^{-2} and 10^{-4} (1% and 0.01%). Clearly, the prediction error of full-trace AET is much smaller than StatStack. 90% of the absolute prediction errors are smaller than 0.17%, while only 55% of StatStack's prediction can reach the same level. The average accuracy improvement of full-trace AET against StatStack in this test is 35.8%. Sampling AET is less accurate than full-trace AET but more accurate than full-trace StatStack. 90% of their prediction errors are smaller than 0.21% and the curves are very close to the full-trace AET curve after 90%. AET sampling is repeated 10 times, and its two CDFs record all the errors, not their average. Figure 7 shows that AET sampling produces stable and accurate results.

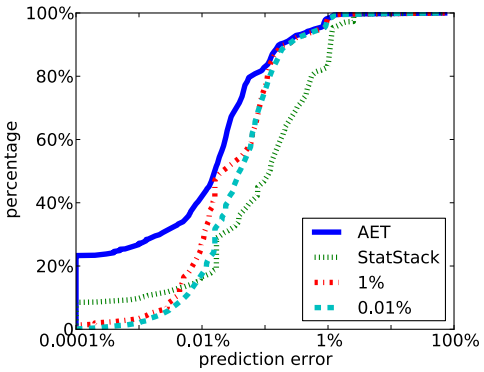


Figure 7: The cumulative distribution function of absolute prediction error

Unlike the AET model using backward reuse time, StatStack uses forward reuse time. It assumes that every reference will have a next reuse. But this is not the case for a trace of a finite length. Every data in the trace has its last reference, and the reuse time of these references are not defined in StatStack. StatStack ignores the impact of these references in its statistical model and characterizes them separately as cold misses. The number of references with no reuse is the same as the working set size. The accuracy of their model is thus influenced by the ratio of the working set size to the trace length.

4.4 Phase Sampling

As mentioned in Section 3.2, phase sampling improves the analysis accuracy for programs with phase behavior. We divide each SPEC CPU2006 benchmark trace into 10 phases of equal length and then use the AET algorithm (full-trace) to profile each phase. Finally, the overall MRC is the average of phase MRCs. In most benchmarks, phase AET sampling is more accurate than non-phase AET sampling. We select four representative benchmarks (`gcc`, `milc`, `wrf`, `hmmer`) and compare phase sampling and non-phase sampling in Figure 8. In these benchmarks, phase analysis leads to significant improvements. The AET models the average eviction time, so it is more accurate when a program shows a steady access behavior. If a program has different phase behavior, we should apply AET analysis on each phase separately as we have done here. More accurate phase analysis may be used to further improve the accuracy of our analysis.

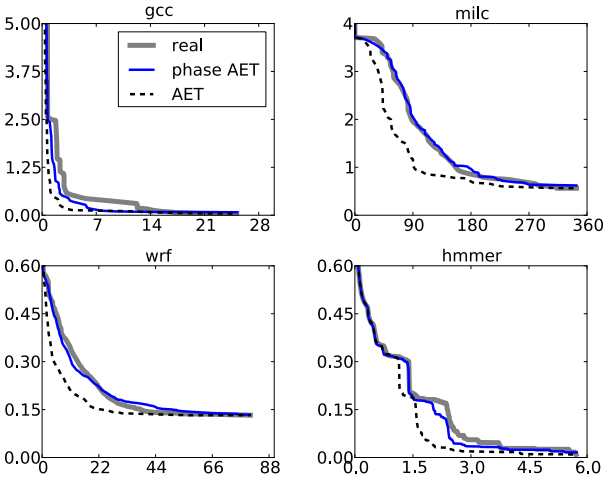


Figure 8: The miss ratio (%) versus cache size (MB) shown for non-phase AET and phase AET

4.5 AET vs ABF Sampling

The footprint-based MRC profiling technique needs recording every access during the monitoring window. The space overhead may be not acceptable for some applications. Wang et al [20]. developed adaptive bursty footprint (ABF) sampling to efficiently measure the footprint of an execution. Extending the design of bursty sampling [35, 36], it approximates the footprint of the entire program by the footprint of small portions. The length of a sampled trace (a burst interval) is bounded by the cache size and minimal miss ratio of interest. The ratio of hibernation and bursty interval is 1000. The miss ratio lower bound is 1%. Therefore, the length of a burst interval was 10^7 to measure 8MB shared cache (131072 cache lines). ABF sampling has several limitations. First, the size of cache is limited by the length of a burst interval. It does not show the MRC for all cache

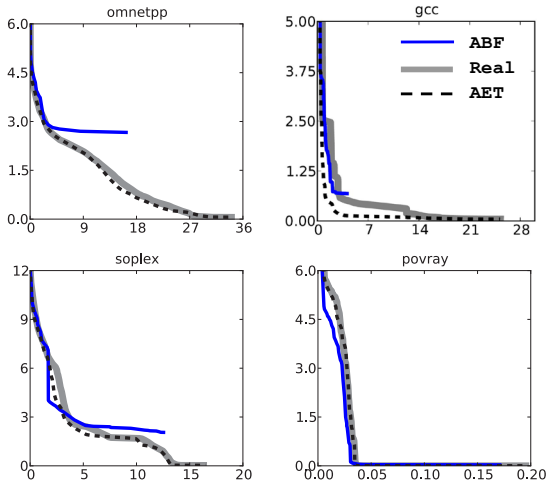


Figure 9: The predicted miss ratio (%) versus cache size (MB) shown for AET sampling and ABF sampling

sizes, unless it measures the complete trace, i.e. no sampling. In comparison, the range of MRC from AET is not influenced by the number of sampled references. Second, an ABF sample is a consecutive portion of a trace. Its result is accurate only if the locality of burst intervals is the same as the locality of the rest (hibernation intervals). In comparison, AET samples cover the entire trace with equal probability.

To evaluate ABF and AET random sampling, we use SPEC CPU2006 benchmarks whose miss ratios are higher than 1%. Due to limitation of space, we only show 4 MRCs (Figure 9) profiled by both techniques with the same sampling rate (1:100). The MRCs of ABF sampling are much shorter than AET sampling, because using bursty interval to represent the entire trace will lose the reference pattern in the hibernation interval. The approximate MRCs of ABF sampling are not as accurate as AET sampling.

4.6 Shared Cache AET

As discussed in Section 2.4, AET is a composable metric and can model shared cache. With the individual AETs of co-run programs, we can predict the MRC of the shared cache they are running on. This technique is essential in task scheduling in a system where shared cache (CPU or storage cache) is deployed. To verify our shared AET modeling technique, we choose four MSR storage traces {prn, src2, web, stg} as a co-run group. They are the traces with the same order of magnitude on length and show totally different patterns in cache usage (see individual MRCs in Figure 10). We assume symmetrical speeds, i.e. equal access rates, to simplify the evaluation, but the extension to asymmetrical cases are straightforward as we showed in Equation 12.

We set the execution length of each trace to be $1.6 * 10^7$, which is the shortest length in the group. With the equal-speed assumption, we generate a combined trace from the four traces. Figure 10 shows the shared cache MRC composed by individual AET modeling of each trace, as well as the real MRC calculated by accurate reuse distance tracking for the combined trace. The shared AET MRC gives a MAE of 0.002, indicating that the shared cache modeling by AET is accurate. The composability of AET is a key advantage over SHARDS and Counter Stacks since these techniques cannot characterize shared cache without co-run testing.

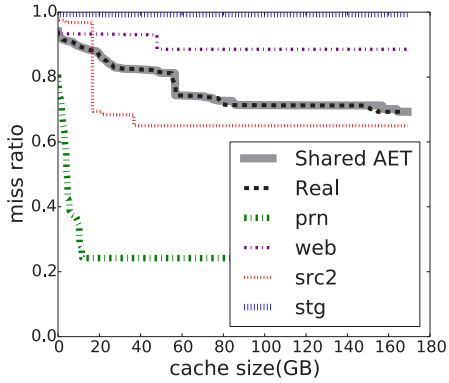


Figure 10: Shared cache MRCs for the combined trace of {prn, src2, web, stg}, as well as the MRCs of four individual traces.

5 Related Work

In 1972, Denning and Schwartz [10] gave a linear-time, iterative formula to compute the average working-set size from reuse times (inter-reference intervals). Mathematically the AET calculation is the same as the average working-set size computed by the Denning-Schwartz formula. In their formulation, Denning and Schwartz assumed infinite traces generated by a stationary process. Later work applied the Denning-Schwartz formula on finite-length traces to compute the average working-set size [37] and LRU stack distance [3]. AET is a new formulation showing that the Denning-Schwartz formula is the solution to AET equations, which are the properties of cache eviction time of all program traces, finite or infinite. Previous work did not address shared cache, which AET can easily model based on eviction-time equalization. Finally, AET is used in sampling analysis of MRC. Sampling was not studied in previous work. However, the previous work modeled arbitrary data size [37, 3] and

Table 3: The space and time complexity of MRC analysis techniques as well as their memory and time consumption measured in master trace

	Time complexity	Space complexity	Memory	Runtime	Composability	Correctness
Stack Processing	$O(NM)$	$O(N)$	10GB	>1 day	No	accurate
Search Tree	$O(N\log M)$	$O(M)$	21GB	482 secs	No	accurate
Scale Tree	$O(N\log\log M)$	$O(M)$	17GB	333 secs	No	bounded err
Footprint	$O(N)$	$O(M)$	17GB	50 secs	Yes	conditional
Counter Stacks	$O(N\log M)$	$O(\log M)$	80MB	1034 secs	No	bounded err
SHARDS	$O(N)$	$O(1)$	2.3MB	29.6 secs	No	conditional
AET model	$O(N)$	$O(1)$	1.7MB	30.5 secs	Yes	conditional

optimal caching policies [3], which we do not consider in this work.

We have started this paper by reviewing the progress of MRC analysis over the last four and half decades. We now give a more comprehensive comparison in Table 3, including the asymptotic complexities, the actual space and time cost (when measuring the merged MSR trace, Section 4.2), the composability (Section 2.4) and correctness properties. AET uses random and reservoir sampling to reduce space cost in practice. In Table 3, the runtime and space overhead of AET for the merged MSR trace is the lowest among all these techniques.

In terms of correctness, Stack Processing [4] and search tree [5, 38] measure reuse distance accurately, and scale tree [39] guarantees the relative precision. Counter Stacks also guarantee an error based on the correctness of Hyperloglog counters. SHARDS uses sampling, and the result is correct if the sampled accesses are representative. The correctness of footprint-based MRC is conditional based on the reuse-window hypothesis [6]. The correctness of AET is conditional as discussed in Section 2.3.

MRC Applications MRC profiling techniques are widely used in different applications. Several studies use on-line MRC analysis for cache partitioning [40, 41], page size selection [25], and memory management [42, 43]. The memory cache prediction [44] also uses on-line MRC detection for storage workload. In high-throughput storage systems, fast MRC tracking is always beneficial.

Our earlier work used footprint-based MRC to optimize memory allocation in the key-value store called Memcached [45]. Previous solutions, e.g. those of Facebook and Twitter, were based on heuristics. We showed that MRC-based optimization was superior in steady-state performance, the speed of convergence, and the ability to adapt to request pattern changes. It achieved over 98% of the theoretical potential. The fast MRC analysis was important since it affects the throughput of Memcached. We used footprint, which was time efficient but consumes a large amount of space (as it is also evident in Table 3). AET sampling should solve the space problem, and it is even faster than footprint.

Fast MRC helps CPU cache optimization. We have developed and evaluated shared cache program symbiosis, which used ABF sampling and footprint composition to co-locate co-run applications to minimize their interference in shared cache [20]. The reuse-distance based techniques in Table 3 are not composable, so they cannot be used in symbiotic optimization. AET is composable, and it can drastically reduce the time and space overhead of shared-cache optimization.

6 Summary

In this work, we present the AET theory, a kinetic model for workload modeling of LRU caches. Using average eviction time (AET) measured by sampling, the AET model consumes liner time and extremely low space for MRC profiling. In our storage workload evaluation AET outperforms Counter Stacks in throughput and space overhead and achieves a comparable performance as SHARDS. At last, We show how AET model can be used to characterize shared cache without co-run testing and with composition invariance.

7 Acknowledgments

The authors wish to thank to Nick Harvey, Carl Waldspurger, Peter Denning, Nohhyun Park, Alexander Garthwaite, Irfan Ahmad and Nisha Talagala for extremely valuable and constructive suggestions for this work and its presentation. This research is supported in part by the National Science Foundation of China (No. 61232008, 61272158, 61328201 and 61472008); the 863 Program of China under Grant No.2015AA015305; the National Science Foundation (No. CNS-1319617, CSR-1422342); a CAS fellowship from IBM and a grant from Huawei.

References

[1] Peter J Denning. Working sets past and present. *Software Engineering, IEEE Transactions on*, (1):64–84, 1980.

[2] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[3] Peter J Denning and Donald R Slutz. Generalized working sets for segment reference strings. *Communications of the ACM*, 21(9):750–759, 1978.

[4] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[5] Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.

[6] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 343–356, 2013.

[7] David Eklov and Erik Hagersten. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 55–65. IEEE, 2010.

[8] Yunlian Jiang, Eddy Z Zhang, Kai Tian, and Xipeng Shen. Is reuse distance applicable to data locality analysis on chip multiprocessors? In *Compiler Construction*, pages 264–282. Springer, 2010.

[9] Xipeng Shen, Jonathan Shaw, Brian Meeker, and Chen Ding. Locality approximation using time. In *ACM SIGPLAN Notices*, volume 42, pages 55–61. ACM, 2007.

[10] Peter J Denning and Stuart C Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, 1972.

[11] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[12] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110. USENIX Association, 2015.

[13] Zachary Drudi, Nicholas JA Harvey, Stephen Ingram, Andrew Warfield, and Jake Wires. Approximating hit rate curves using streaming algorithms. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 40. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[14] Peter J Denning, Craig H Martell, and Vint Cerf. *Great Principles of Computing*. MIT Press, 2015.

[15] Chen Ding, Xiaoya Xiang, Bin Bao, Hao Luo, Ying-Wei Luo, and Xiao-Lin Wang. Performance metrics and models for shared cache. *Journal of Computer Science and Technology*, 29(4):692–712, 2014.

[16] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multiprocessor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340–351. IEEE, 2005.

[17] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *25th Anniversary International Conference on Supercomputing Anniversary Volume*, pages 323–334. ACM, 2014.

[18] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul M. Chilimbi. All-window profiling and composable models of cache sharing. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 91–102, 2011.

[19] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360. IEEE, 2011.

[20] Xiaolin Wang, Yechen Li, Yingwei Luo, Xiameng Hu, Jacob Brock, Chen Ding, and Zhenlin Wang. Optimal footprint symbiosis in shared cache. In *CCGRID*, 2015.

[21] Jacob Brock, Yechen Li, Chencheng Ye, and Chen Ding. Optimal cache partition-sharing : Dont ever take a fence down until you know why it was put up. robert frost. In *International Conference on Parallel Processing*, 2015.

[22] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the International Symposium on Memory Management*, pages 91–100, 2008.

[23] Derek L. Schuff, Milind Kulkarni, and Vijay S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 53–64, 2010.

[24] David K. Tam, Reza Azimi, Livio Soares, and Michael Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–132, 2009.

[25] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 339–349, 2005.

[26] Kristof Beyls and Erik H. D'Hollander. Discovery of locality-improving refactoring by reuse path analysis. In *Proceedings of High Performance Computing and Communications. Springer. Lecture Notes in Computer Science*, volume 4208, pages 220–229, 2006.

[27] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[28] Éric Fusy, G Olivier, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *In AofA07: Proceedings of the 2007 International Conference on Analysis of Algorithms*, 2007.

[29] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[30] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *FAST*, volume 3, pages 115–130, 2003.

[31] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 169–180. ACM, 2005.

[32] Erik Berg and Erik Hagersten. Statcache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pages 20–27. IEEE, 2004.

[33] cpu2006. https://www.spec.org/cpu2006/, 2015. [Online].

[34] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[35] Matthew Arnold and Barbara G Ryder. A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179, 2001.

[36] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. Pacer: proportional detection of data races. *ACM SIGPLAN Notices*, 45(6):255–268, 2010.

[37] Donald R. Slutz and Irving L. Traiger. A note on the calculation working set size. *Communications of the ACM*.

[38] George Almasi, Calin Cascaval, and David A. Padua. Calculating stack distances efficiently. In *Proceedings of the ACM SIGPLAN Workshop on Memory System Performance*, pages 37–43, Berlin, Germany, June 2002.

[39] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.

[40] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, pages 1–12. ACM, 2001.

[41] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[42] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 177–188. ACM, 2004.

[43] Yul H. Kim, Mark D. Hill, and David A. Wood. Implementing stack simulation for highly-associative memories. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 212–213, 1991.

[44] Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[45] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX ATC*, 2015.

Scalable In-Memory Transaction Processing with HTM

Yingjun Wu and Kian-Lee Tan
School of Computing, National University of Singapore

ABSTRACT

We propose a new HTM-assisted concurrency control protocol, called HTCC, that achieves high scalability and robustness when processing OLTP workloads. HTCC attains its goal using a two-pronged strategy that exploits the strengths of HTM. First, it distinguishes between hot and cold records, and deals with each type differently – while accesses to highly contended data are protected using conventional fine-grained locks, accesses to cold data are HTM-guarded. This remarkably reduces the database transaction abort rate and exploits HTM's effectiveness in executing low-contention critical sections. Second, to minimize the overhead inherited from successive restarts of aborted database transactions, HTCC caches the internal execution states of a transaction for performing *delta-restoration*, which partially updates the maintained read/write set and bypasses redundant index lookups during transaction re-execution at best effort. This approach is greatly facilitated by HTM's speedy hardware mechanism for ensuring atomicity and isolation. We evaluated HTCC in a main-memory database prototype running on a 4 socket machine (40 cores in total), and confirmed that HTCC can scale near-linearly, yielding high transaction rate even under highly contended workloads.

1 INTRODUCTION

With the introduction of Intel's newly released Haswell processors, hardware transactional memory (HTM) is finally available in mainstream computing machines. As promised in its initial proposal twenty years ago, HTM greatly simplifies the implementation of correct and efficient concurrent programs.

A major target application for exploiting HTM is modern multicore OLTP databases, where sophisticated concurrency control protocols must be designed to guarantee isolation among threads. To benefit from HTM, effective solutions should reduce HTM's high abort rate caused by capacity overflow, which is a major limitation of Intel's current implementation. A promising approach [34] is to apply HTM to optimistic concurrency control (OCC) protocol, where transaction computation is entirely detached from commitment. While achieving high transaction rate under certain scenarios, this solution unfortunately gives rise to unsatisfactory performance when processing highly contended workloads. This is because conventional OCC is intrinsically sensitive to skewed data accesses, and, in fact, protecting OCC's commitment with the speculative HTM can make the protocol even more vulnerable to contentions.

In this paper, we introduce HTCC, a new HTM-assisted concurrency control mechanism that achieves robust transaction processing even under highly contended workloads. The design of HTCC is inspired by two observations. First, massive volumes of transactions in a contended workload skew their accesses on a very few popular data records, and optimistically accessing these records are likely to result in high abort rate due to data conflicts. Second, an aborted database transaction may be successively re-executed before it is eventually committed, and such re-executions incur the overhead of repeatedly retrieving the same records via index lookups.

Based on these facts, HTCC adopts a two-pronged approach that takes full advantage of the strengths of HTM. On the one hand, HTCC splits the data into hot and cold records, and a combination of pessimistic fine-grained locking and optimistic HTM is leveraged to protect a transaction's accesses differentially according to the degree of contentions. This mechanism avoids frequent data conflicts on highly contended records, and fully exploits HTM's effectiveness in executing lowly contended critical sections. On the other hand, to minimize the overhead incurred from successive restarts of aborted database transactions, HTCC further maintains a thread-local structure, called *workset cache*, to buffer the accessed records of each operation within a transaction. When an aborted transaction is re-executed, HTCC per-

forms an efficient *delta-restoration*: records to be read or written during the re-execution are fetched directly from the cache, and unnecessary overhead caused by redundant index lookups is largely reduced. However, this design requires maintenance of a transaction's read/write sets and handling of deadlocks. With HTM, potential deadlocks are now handled by HTM's hardware scheme; this not only simplifies the design but also minimizes the overhead of deadlock resolution (as there is no longer the need to manage deadlocks for cold data explicitly).

We implemented `HTCC` in CAVALIA (source code: `https://github.com/Cavalia/Cavalia`), a main-memory database prototype that is built from the ground up. Our evaluation confirmed that `HTCC` can yield much higher transaction rate on a 4 socket machine (40 cores in total) compared to the state-of-the-art concurrency control protocols, especially under highly contended workloads.

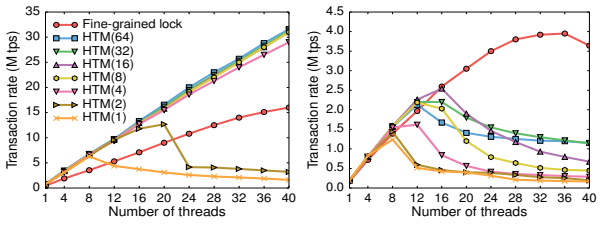
This paper is organized as follows: Section 2 provides a preliminary study to show the design intuition of `HTCC`. Section 3 explains how `HTCC` reduces database transaction abort rate caused by data conflicts, and Section 4 describes how `HTCC` eliminates unnecessary overhead in restarting aborted database transactions. We report extensive experiment results in Section 5. Section 6 reviews related works and Section 7 concludes this work.

2 PRELIMINARY

Hardware transactional memory. Hardware transactional memory (HTM) aims at simplifying the implementation of correct and efficient parallel algorithms. With the release of Intel's Transactional Synchronization Extension (TSX) instructions, programmers nowadays can simply use `xbegin` and `xend` to guard a code segment that should be executed transactionally. Upon finishing an HTM region, all the memory writes encapsulated in this region will appear atomically. TSX instructions also eliminate the need for implementing software mechanisms for deadlock resolution, and its intrinsic hardware-level mechanism efficiently ensures atomicity and isolation with little overhead.

The TSX instructions use the CPU cache to store execution buffers for an HTM region. The cache-coherence protocol is leveraged to track data conflicts at cache-line granularity. Such use of existing features in modern CPU architectures makes it possible to provide low-overhead hardware support for transactional memory. To guarantee forward progress, a fallback routine that acquires coarse-grained exclusive locks will be executed after an HTM region has been aborted for a certain predetermined number of times. We refer to this number as the *restart threshold*.

The major limitation of the current HTM implementation is its high abort rate, which is caused by either capacity overflow or data conflicts. First, the capacity of an HTM region is strictly constrained by both the CPU cache size and the cache associativity, and any violation of such constraint can directly give rise to HTM region abort. Second, HTM executes the protected critical sections in a speculative manner, and two concurrent HTM regions updating to the same memory address can result in either region being aborted. While the capacity-overflow problem can be addressed through a careful design of critical sections, data conflicts among dozens of threads can severely restrict the HTM scalability, making it less effective as a general mechanism to support a wide spectrum of concurrent program workloads. Besides the costly restart overhead, frequent aborts severely hurt HTM's performance because of the well-known *lemming effect* [8]. As the repeated aborts of a single HTM region eventually lead to the acquisition of a global coarse-grained lock, all other HTM regions cannot proceed until the lock is released. This consequently forces all the concurrent HTM regions to abort, resulting in fully serialized execution without parallelism.



(a) Low-contention workload. (b) High-contention workload.

Figure 1: Processing multi-key transactions. Please note the difference in y-axis ranges.

Figure 1 compares HTM with conventional fine-grained locking mechanism when processing multi-key transactions on a standard hash map with 1 million data records. Each transaction randomly updates 10 records, and the access contention is controlled by a parameter θ, indicating the skewness of the Zipfian distribution. The fine-grained locking scheme acquires locks associated with all the targeted records and prevents deadlocks by pre-ordering its write sets; such an approach is also used in the validation phase of traditional OCC protocol [32]. The restart threshold for HTM regions is varied from 1 (see `HTM(1)`) to 64 (see `HTM(64)`). Figure 1a shows that, by setting the restart threshold to larger than 4, HTM achieves 2X better performance (31 vs 16 M tps) when processing low-contention workloads ($\theta = 0$) with 40 threads. However, the results exhibited in Figure 1b indicate that the transaction rate achieved by HTM deteriorates significantly under highly contended work-

loads ($\theta = 0.8$). In particular, with restart threshold set to 64, HTM reaches a transaction rate of 1.1 M tps with 40 threads enabled. This number is remarkably smaller than 3.7 M tps, which is attained by fine-grained locking.

Given the experimental results presented above, we confirm that Intel's HTM implementation is more suitable for protecting lowly contended data accesses. Fine-grained locking, in comparison, is still more attractive for executing high-contention workloads.

Optimistic concurrency control. Modern multicore OLTP database is a major target for applying HTM. However, the capacity limit of Intel's HTM implementation prevents a one-to-one mapping from database transactions to HTM regions. A promising solution [34] to address this problem is to apply HTM to conventional OCC protocol [15], which splits the execution of a database transaction into three phases: (1) a *read* phase, which executes all the read and write operations in the transaction according to the program logic without any blocking; (2) a *validation* phase, which checks the transaction conflicts by certifying the validity of the transaction's read set; and (3) a *write* phase, which installs all the transaction writes to the database atomically. Fine-grained locks for all the records that are accessed by the transaction must be acquired on entering the validation phase and be released when terminating the write phase. A transaction T_1 will be aborted if it fails the validation phase. Such a failure indicates that a certain committed concurrent transaction T_2 has modified the values of certain records in T_1's read set during T_1's read phase. We refer to the time period during T_1's read phase as *vulnerable window*, where inconsistent accesses can occur.

To improve the performance of conventional OCC, existing solution [34] adopts HTM to completely replace the fine-grained locks that are used during the transaction's validation and write phases. However, as HTM executes guarded code segment speculatively, the validation and the write phases consequently become vulnerable to contention. More specifically, any committed updates from a concurrent transaction T_2 can abort a transaction T_1 even if T_1 is within the validation and the write phases that are protected by HTM region, as HTM does not forbid any concurrent accesses. As a result, the vulnerable window in such a HTM-assisted OCC protocol can span across the whole database transaction. Figure 2 depicts the difference between conventional OCC [15] and existing HTM-assisted OCC [34].

The characteristic of the existing HTM-assisted OCC protocol makes it unattractive for processing highly contended workloads. Therefore, we design a new concurrency control protocol that fully exploits the power of HTM to achieve robust transaction processing.

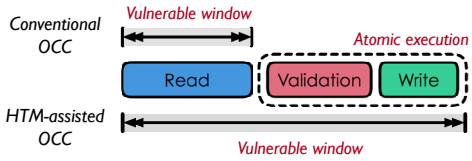


Figure 2: Conventional OCC v.s. HTM-assisted OCC.

3 REDUCING ABORT RATE

HTCC is an HTM-assisted OCC-style protocol that can achieve high transaction rate by leveraging a combination of fine-grained locking and HTM. The key intuitions are that contended OLTP workloads skew their accesses on a small portion of data records in the database, and HTM yields a remarkably higher performance when protecting low-contention critical sections. This section describes how HTCC fully exploits the benefits of HTM and reduces the likelihood of database transaction aborts.

3.1 Data Record Classification

HTCC splits the data into hot and cold records so as to choose the best scheme for locking each single record that is accessed by a certain database transaction. As depicted in Figure 3, HTCC maintains five metadata fields for each data record: (1) a *lock* flag showing the record's lock status; (2) a *ts* attribute recording the commit timestamp of the last writer; (3) a *vis* flag indicating whether the record is visible to inflight transactions; (4) a *hot* flag specifying whether the record is highly contended; and (5) a *cnt* attribute counting the number of validation failures triggered by the conflicting accesses to the record during a certain time slot.

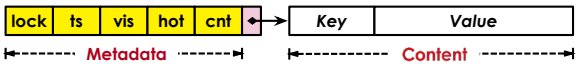


Figure 3: Structure of a data record.

The *vis* flag of a record has three possible states: public state indicating that the record is visible to all the inflight transactions; private state indicating that the record is newly inserted by an uncommitted transaction; and dead state indicating that the corresponding record has been deleted by a committed transaction. During transaction processing, HTCC automatically tracks a transaction's conflicting accesses and atomically increments the *cnt* field for the record to which the access raises the validation failure. To achieve this, the *cnt* field is packed into a single word to allow CAS operation. A background thread is spawned to periodically (e.g., 60 seconds) check the failure counts and mark the most contended records by flipping the associated *hot* flag. A

record that is no longer frequently accessed during a constant time duration will be tagged back to cold. The flip of a *hot* flag is performed transactionally, and every in-flight transaction will always see a consistent *hot* flag of each record from its start to its final commitment. That is, the background thread can change a record's *hot* flag only if no concurrent transaction is accessing this record.

3.2 Hybrid Transaction Processing

Like OCC, HTCC divides the execution of a transaction into read, validation, and write phases. However, HTCC adopts a hybrid synchronization mechanism to execute a single transaction: on the one hand, pessimistic fine-grained locking is employed to protect the accesses to hot records; on the other hand, HTM region is leveraged to guard the accesses to cold records. For each transaction, HTCC maintains a thread-local read/write set (rw-set) to store information about the records read or written by the transaction. Each element in the rw-set keeps an array with five basic fields: $\langle record, data, ts, hot, type \rangle$, where *record* stores the pointer to the original record in the database, *data* buffers a local copy of the accessed record's content, *ts* identifies the timestamp value of the record at the time it was accessed by the transaction, *hot* indicates whether the accessed record is contended or not, and *type* shows the access type (read (R), write (W), or both (RW)) to the record. When accessing a hot record in a transaction, HTCC performs updates directly to the original record in the database, and the locally buffered copy is used for content recovery should the database transaction be aborted. However, when accessing a cold record, updates performed by the transaction will be temporarily held in the local copies before they are being installed to the original records. The following subsections describe the three phases of HTCC in detail.

Read phase. HTCC speculatively accesses cold records during a transaction's read phase. The function `AccessRecord(r, T)` in Figure 4 depicts how an operation with access type T on record r is performed in the read phase. To access a record r in a transaction, HTCC first checks r's *hot* flag and acquires the associated read or write lock if this flag is set to true. Subsequently, a new element e in the rw-set is created for r. All the fields, including *record*, *data*, *ts*, *hot*, and *type* are atomically stored in e. The *type* field in e is updated to T to indicate the access type (i.e., read, write, or both).

The acquisition of fine-grained locks for hot records during the read phase can cause deadlocks. Therefore, certain deadlock-resolution scheme must be employed to guarantee forward progress. Our current implementation adopts a no-wait-style strategy [5], which aborts a database transaction if a certain lock cannot be obtained within a certain time range.

```
AccessRecord(r, T):
  // ========== LOCK HOT RECORDS ==========
  if (r.hot == true) then
    r.acquire_lock();
  e = rw_set.insert(r);
  e.set(r, r.data, r.ts, r.hot, T);

Commit():                                   RTM-guarded region
  //  ============ VALIDATION ==============
  do
    is_success = true;

    HTM_BEGIN(); // begin HTM txn
    foreach e in rw_set do
      if e.hot == false then
        if e.type == R || e.type == RW then
          if e.ts != e.record.ts then
            HTM_END(); // end HTM txn
            Repair transaction;
            is_success = false;
            break;
  while (is_success == false);

  commit_ts = generate_commit_ts();

  // ======== UPDATE COLD RECORDS ==========
  foreach e in rw_set do
    if e.hot == false then
      if e.type == W || e.type == RW then
        install(e.record, e.data);
        e.record.ts = commit_ts;
  HTM_END(); // end HTM txn

  // ======== UPDATE HOT RECORDS ==========
  foreach e in rw_set do
    if e.hot == true then
      if e.type == W || e.type == RW then
        install(e.record, e.data);
        e.record.ts = commit_ts;

  // ======== UNLOCK HOT RECORDS ==========
  foreach e in rw_set do
    if e.hot == true then
      e.record.release_lock();
```

Figure 4: The framework of HTCC.

Validation phase. As HTCC touches cold records in a consistency-oblivious manner through the read phase, a validation phase must be invoked subsequently to check the validity of every cold record that is read by this transaction. Similar with the existing HTM-assisted OCC protocol [34], HTCC leverages HTM region to protect the atomic execution of the validation phase.

As shown in the function `Commit()` in Figure 4, HTCC begins its validation phase by informing the hardware to start executing in transactional mode with `xbegin` instruction. After that, HTCC iterates over the rw-set S and checks if any cold record read by the current transaction has been modified by any committed concurrent transaction. The record value loaded at the read phase is determined to be *invalid* if the corresponding element e in S stores a timestamp ts that is different from that attached in the original record (i.e., $e.record.ts$).

Such an invalidation is made by the *inconsistent* read operation that first touches this record. On encountering any invalid element in the rw-set, HTCC will *repair* the transaction instead of re-executing it from scratch. We defer the discussion on the repairing process to the next section. The validation phase terminates when the transaction eventually reads all the records consistently or aborts due to deadlocks (for hot records).

Write phase. A transaction that has passed the validation phase will enter the write phase to make every update visible to all the concurrent transactions. The write phase of HTCC is also described in Figure 4. Under the protection of HTM region, HTCC installs the locally buffered contents back to every accessed cold record, with a commit timestamp *commit_ts* attached to identify the partial dependencies among transactions [32]. In particular, pointer swap [34] is used for local-copy installation, as this mechanism minimizes the HTM region capacity. On exiting the HTM region, the timestamp *commit_ts* is written to all the updated hot records, and fine-grained locks are finally released to make every hot record visible to all other inflight transactions.

The protocol described above leverages the utmost of two distinct synchronization mechanisms: accesses to hot records are protected by fine-grained locking, which yields satisfactory performance when processing high-contention workloads; accesses to cold records are guarded by HTM region, which achieves much higher throughput when executing lowly contended workloads. The next section continues to explore the feature of HTM that further accelerates HTCC's performance.

4 MINIMIZING RESTART OVERHEAD

While the proposed hybrid processing mechanism can effectively reduce the database transaction abort rate, it may still result in expensive re-execution overhead if an inflight database transaction is blindly rejected once validation failure occurs. We develop a *transaction repair* mechanism that greatly benefits from HTM's guarantee of atomicity and isolation. By keeping track of the accesses of each operation at runtime in a *workset cache*, HTCC re-executes a failed transaction only by performing *delta-restoration*, which effectively reutilizes the contents in the cache to speed up the restoration of each operation in the transaction. Thanks to the use of HTM, the accesses to cold records during the repair stage are still processed optimistically, bringing little overhead to the system runtime.

4.1 Workset Cache

The processing cost of a database transaction is dominated by the time to retrieve data records from the corresponding database indexes. On detecting an invalid element in the rw-set during validation, a database transaction has to be (successively) aborted and re-executed from scratch, fetching the same records through index lookups multiple times. The overhead caused by these redundant index lookups can be largely reduced by caching the accessed records in thread-local data structures during the initial execution of a transaction.

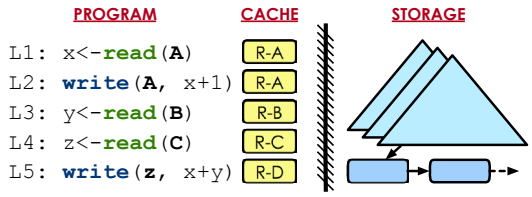


Figure 5: Workset cache for a transaction.

Based on this intuition, HTCC maintains a *workset cache*, which is constructed during the read phase, that buffers a list of record pointers for all the records that are accessed by each operation in a transaction. In particular, we refer to the set of records accessed by a certain operation as the operation's *workset*. Figure 5 shows such a caching structure maintained for a running transaction. The initial execution of the read operation L_1 fetches a data record *R-A* using candidate key *A* and caches the pointer to *R-A* in the corresponding entry of the workset cache. Similarly, the write operation L_5 loads the record *R-D* through the index and buffers its pointer into the workset cache before performing real updates. Should the database transaction be aborted, the subsequent re-execution can tap on the record pointers from the workset cache to minimize index traversal. This approach saves a significant amount of efforts for committing transactions that are prone to abort. Note that for a range operation that accesses multiple records, its workset needs to maintain all the pointers to its targeted records.

4.2 Delta-Restoration

With the workset cache maintained for each transaction, HTCC repairs an aborted database transaction with *delta-restoration*, which re-utilizes the internal states created during the transaction execution without restarting the transaction from scratch. The main idea of delta-restoration is that, by leveraging the maintained workset cache, most of the index lookups can be bypassed by directly accessing the buffered record pointers, and only a small portion of the re-set needs to be updated, and the incurred overhead to ensure atomicity and isolation can be largely reduced through HTM's support.

As shown in Figure 4, on confronting a validation failure of a transaction, HTCC directly exits the HTM region

and starts *repairing* the entire transaction, during which stage all the operations in a transaction will be *restored* at best effort. Figure 6 describes how HTCC restores an operation *opt* with access type *T* (i.e., read, write, or both) in a transaction with two distinct schemes: *fast-restoration* and *slow-restoration*.

```
RESTORE(opt, T):
  if is_key_invariant(opt) == false then
  // ========== FAST-RESTORATION ==========
    W = get_workset(opt);
    perform(opt, W); // perform read or write
  else
  // ========== SLOW-RESTORATION ==========
    V = index_lookup(opt);
    W = get_workset(opt);
    set_workset(opt, V);
                                  Update rw-set
    ins_set, del_set = compute_delta(V, W);
    foreach del in del_set do
      if del.hot == true then
        del.release_lock();
      rw_set.erase(del);
    foreach ins in ins_set do
      if ins.hot == true then
        ins.acquire_lock(); // deadlock?
      e = rw_set.insert(ins);
      e.set(ins, ins.data, ins.ts, ins.hot, T);

    perform(opt, V); // perform read or write
```

Figure 6: Restore a single operation in a transaction.

Fast-restoration. This scheme re-executes an operation in the transaction without invoking any index lookups. Instead, all the targeted records are directly fetched from the workset cache constructed at runtime. In particular, fast-restoration only works for an operation *opt* if the accessing key has not been updated, which essentially indicates that the corresponding records to be accessed by this operation should be identical to those maintained in the workset cache.

Slow-restoration. This scheme is specifically designed for an operation whose accessing key has been changed due to the restoration of its parent operations. Compared with fast-restoration, slow-restoring an operation *opt* is more expensive, as the targeted records that should be accessed by *opt* have to be re-fetched through index lookup. Consequently, the corresponding workset maintained for *opt* in the caching structure must also be refreshed. As shown in Figure 6, after retrieving all the records required by *opt* through index lookup using a candidate key *key*, the slow-restoration refreshes its workset cache by updating the *opt*'s workset from *W* to *V*. Subsequently, the difference between *W* and *V* is computed. In particular, the delta-set *ins_set* contains all the new records that must be inserted into the rw-set, and the delta-set *del_set* contains all the records that should be removed from the rw-set.

Figure 7 illustrates the two types of restoration mechanisms using the sample transaction shown in Figure 5. During the repair of the transaction, the restoration of L_2 directly fetches the pointer *R-A* from its local cache without resorting to the index lookup. However, the accessing key of L_5 is dependent on the output of L_4, which can be modified during L_4's restoration. Once a change in accessing key is detected, the restoration of L_5 has to traverse the index and update the workset before performing the real write.

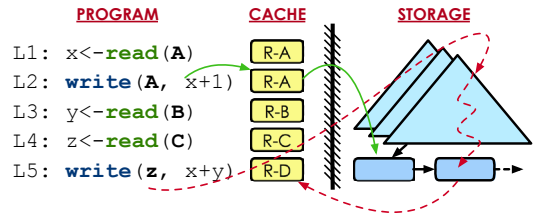


Figure 7: Fast-restoration (solid line) and slow-restoration (dashed line) of an operation.

Delta-restoration, however, is not for free, as the update of the rw-set has to resort to certain synchronization mechanism for protecting records that are newly inserted into the rw-set. In particular, deadlock can occur if fine-grained locking is utilized. HTM's deadlock-free scheme to protect complex critical sections in contrast provides an elegant and efficient way for tackling this problem. Throughout the repair stage, HTCC keeps processing cold records in a consistency-oblivious manner, meaning that no fine-grained locks are acquired to guarantee atomicity. The consistency checking of these newly inserted cold records are deferred until the next round of validation, where HTM region is leveraged to guard the critical section. This design effectively offloads the overhead brought by deadlock resolution to the hardware mechanism, and the software fine-grained locking mechanism is only used for protecting hot records. When updating the rw-set, all the locks associated with the hot records maintained in the delta-set *del_set* should be released during slow-restoration. Similarly, every newly fetched hot record in *ins_set* must be locked before adding them into the rw-set *rw_set*.

HTCC returns back to the validation phase once all the operations within a transaction has been restored. It should be noted that for transactions containing conditional branches, some operations may no longer be executed due to the change of decisions made in those conditional statements. As a result, HTCC further needs to remove all the elements that are no longer accessed by the transaction from the rw-set before validating it again. A transaction can proceed to commit only if all the elements in the rw-set have been successfully validated.

To conclude, the use of HTM has contributed to the efficiency of this repair mechanism and ultimately the superiority of HTCC. The gain in performance comes from two sources. First, there is no need to maintain locks

for cold data. This significantly reduces the overhead of the repair mechanism as the number of cold data is much larger than that of hot data. Second, the hardware support provided for HTM to ensure atomicity and isolation of the HTM regions (the part of the transactions that access the cold data) naturally offers superior performance (as compared to a software-based scheme).

4.3 Operation Dependency Extraction

While workset caching can reduce the overhead of transaction re-execution, an invalidation failure will trigger restoration of all the operations within the transaction. As an inconsistent read usually affects only a limited portion of the transaction program, performance can be improved if the code segment to be re-executed can be minimized. In fact, an understanding of the dependencies across operations can determine the subset of operations to be restored when invalidation occurs. HTCC therefore extracts two types of dependencies from the program at compile time: (1) *key dependency*, which occurs when the accessing key of an operation depends on the result of its previous operation; and (2) *value dependency*, which occurs when the non-key value to be used by an operation depends on the result of its previous operation. Figure 8 depicts the dependency relations within the transaction shown in Figure 5. Operations L_2 and L_5 are value-dependent on L_1, as L_1 assigns the variable x that will be used as non-key value in both L_2 and L_5. Operation L_5 is also key-dependent on its previous read operation L_4, which generates the accessing key z for L_5.

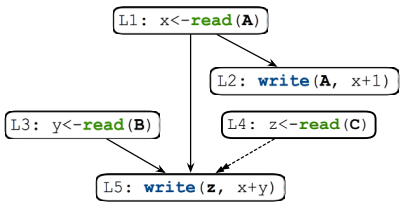


Figure 8: Dependency relations within a transaction.

To identify the read operation that produces the invalid element in the rw-set, HTCC adds an *operation* field to each element. Such a field tracks the operation that first reads the corresponding record, and will be further used for identifying all the affected operations that must be re-executed for preserving correctness.

Figure 9 shows how the extracted dependencies facilitates the repair of a transaction. Given an invalid element e that causes the validation failure, the repair mechanism starts by identifying the operation *opt* that performs the first read to e. Since *opt* is the initial inconsistent operation, all its descendant operations must be restored to guarantee serializability. Correct restoration mechanism

```
REPAIR(e):
  opt = e.operation;
  Restore(opt); // restore operation
  DS = extract_dependent_set(opt);
  PS = DS;
  while PS.is_empty() == false do
    opt = PS.pop_front();
    Restore(opt); // restore operation
    RS = extract_dependent_set(opt);
    PS.append_set(RS);
```

Figure 9: Repair a transaction with dependency relations.

is selected for each inconsistent operation by checking its dependency relations with its parent operations. After restoring an operation *opt*, all its child operations in the dependent-set *RS* will be inserted into the restoration set *PS*. The repair mechanism terminates once all the potentially inconsistent operations have been restored. Subsequently, HTCC restarts the validation phase, which eventually terminates if all the elements in the transaction's read set pass the validation phase.

In the current version of our system, all the transactions are implemented in C++. We leveraged LLVM Pass framework [1] to extract dependencies from a transaction program at compile time. While the classic interprocedural data-flow analysis techniques [27] can precisely capture operation relations for programs containing function calls, we currently do not yet support dependency extraction for nested transactions, where a transaction can be invoked in the process of another transaction. We leave the support of nested transactions as a future work.

4.4 Extended Database Operations

This section depicts how HTCC supports inserts, deletes, and range queries without sacrificing serializability.

Inserts. To insert a record R in a transaction T_1, HTCC performs an early insertion to hold a place for R in the database during T_1's read phase. The to-be-inserted record R is added to T_1's rw-set, with its *vis* flag set to `private`, indicating that R is not yet ready for access. The *vis* flag is updated to `public` only if T_1 has been successfully committed. If a concurrent transaction T_2 attempts to perform the same insertion, only one transaction can succeed. Should T_1 commit before T_2, T_2 will detect this conflict during the validation phase and trigger repair mechanism to handle failed insertion.

Deletes. A transaction marks the deletion of an existing record R by setting R's *vis* flag to `dead`. A garbage collection thread periodically reclaims all the deleted records that are still maintained in the database. A deleted record R can be safely reclaimed if R is no longer referred to in any running transaction's rw-set.

Range queries. Range queries can lead to phantom problem [9]. Instead of resorting to the classic *next-*

key locking mechanism [23] for tackling such problem, HTCC leverages a solution that is first introduced by Silo [32]. HTCC maintains both the version number and the leaf pointers in a transaction's rw-set. Any structural modification caused by concurrent inserts or deletes will be detected by checking the version's validity. Once an invalidity is discovered, HTCC restores any inconsistent operations by invoking the repair mechanism.

5 SERIALIZABILITY

In this section, we give an informal argument on why HTCC is capable to enforce full serializability.

First, HTCC's a combination of fine-grained locking and HTM yields serializable results. We reduce the proposed scheme to two-phase locking (2PL) protocol. Let us consider a transaction T accessing two records R_1 and R_2. If R_1 and R_2 are both tagged as hot records, then HTCC locks both records using fine-grained locking scheme, which is equivalent to 2PL, hence database serializability is guaranteed. If R_1 and R_2 are both tagged as cold records, then HTCC leverages HTM region to protect the atomic access to them. This scheme is serializable, as is proved in the previous work [34]. If R_1 is a hot record and R_2 is a cold record, then HTCC acquires the lock on R_1 before entering the HTM region where R_2's atomic access is protected. On passing the validation, HTCC exits the HTM region and eventually releases the lock on R_1. During this stage, no new lock is acquired, and hence the protocol obeys the constraints set in the 2PL scheme. Therefore, this protocol is serializable.

Second, HTCC's repair mechanism in the validation phase does not compromise database serializability. HTCC executes the repair mechanism in a consistency-oblivious manner, that is, no synchronization schemes are applied during this stage. However, once the repair mechanism completes, HTCC falls back to the very beginning of the validation phase, and all the records read by the transaction will be validated within an HTM region. Hence, the serializability is guaranteed.

To conclude, HTCC yields serializable results when processing transactions.

6 EXPERIMENTS

In this section, we evaluate the performance of HTCC. All the experiments are performed in a main-memory database prototype, called CAVALIA, that is implemented from the ground up in C++.

We deployed CAVALIA on a multicore machine running Ubuntu 14.04 with four 10-core Intel Xeon Processor E7-4820 clocked at 1.9 GHz, yielding a total of 40 cores. Each core owns a private 32 KB L1 cache and a private 256 KB L2 cache. Every 10 cores share a 25 MB L3 cache and a 64 GB local DRAM. We have enabled hyper-threading, and the machine provides 80 hardware threads in total. Note that due to hyper-threading, the experiments with more than 40 threads may yield sub-linear scalability, as we shall see later in this section. We compare HTCC with a collection of concurrency control protocols, as listed below:

2PL: The classic two-phase locking with wait-die strategy adopted for resolving deadlocks [5].

OCC: The classic OCC protocol with scalable timestamp generation mechanism applied [32].

SOCC: A variation of conventional OCC protocol implemented in Silo [32].

HOCC. An HTM-assisted OCC protocol proposed by Wang et al [34].

HTO. An HTM-assisted timestamp ordering protocol proposed by Leis et al [17].

All the protocols in comparison are implemented in CAVALIA. Similar with the original implementation of Silo, our SOCC implementation also adopts the epoch-based timestamp generation mechanism and eliminates the need of tracking anti-dependency relations. However, instead of using Masstree [22] as the underlying index, we leveraged libcuckoo [19] for indexing primary keys. The implementation of HOCC is adapted from that of OCC. Following the description of Wang et al. [34], we directly protected the validation phase using HTM region and adopted pointer swap to minimize the likelihood of hardware transaction abort caused by capacity overflow. Our implementation of HTO basically follows the original paper [17], but we ignored the proposed storage-layer optimizations, as the storage layout in a database system should be orthogonal to the upper-layer concurrency control protocol. We use the TPC-C benchmark [2] to evaluate the performance.The workload contention is controlled by changing the number of warehouses, and decreasing this number can increase the contention. In our experiments, HTCC's automatic scheme (see Section 3.1) is applied for identifying hot records.

Existing bottlenecks. We begin our experiments with a detailed analysis on the state-of-the-art HTM-assisted concurrency control protocol [34]. We first measure the performance of HOCC to see whether this protocol is robust to workload contentions. Figure 10 shows the transaction rate achieved by the HOCC protocol with the HTM region's restart threshold increased from 1 (HOCC(1)) to 64 (HOCC(64)). The number of warehouses is set to 40, yielding a low data contention. As the result indicates, with the restart threshold set to 64, HOCC can successfully scale on 80 threads, achieving a transaction rate at over 2.1 M transactions per second (tps). This number is 15% higher than that achieved by OCC, showing that

HTM can provide a more efficient scheme for executing low-contention critical sections.

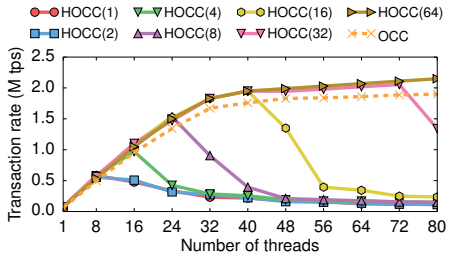


Figure 10: Performance of HOCC with different restart thresholds. The number of warehouses is set to 40.

While the previous work [34] as well as our experiment shown above confirmed the efficiency of the existing HTM-assisted protocol under low-contention workloads, the performance can drop drastically when processing highly contended workloads. Figure 11 shows the experiment result. By setting the number of warehouses to 4, the performance of HOCC deteriorates drastically regardless of the restart threshold. In particular, with the thread count set to 80, HOCC only yields around 200 K tps, while OCC under the same scenario achieves over 550 K tps, exhibiting a much better scalability. This is mainly because the speculative execution of HTM can give raise to high abort rate, and frequent aborts subsequently result in costly restart overhead as well as the lemming effect, which eventually leads to pure serialized execution without parallelism. This result is also consistent with that reported by Makreshanski et al [21].

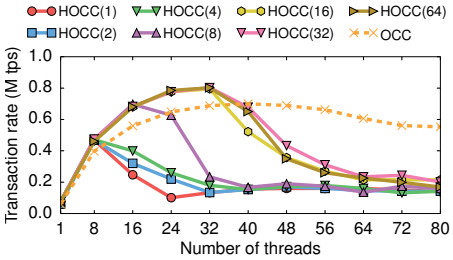


Figure 11: Performance of HOCC with different restart thresholds. The number of warehouses is set to 4.

An interesting observation in the experiment above is that, as shown in Figure 12, the database transaction abort rate[1] generated by HOCC drops with the increase of thread count. With the restart threshold set to 64, HOCC yields an abort rate of 0.45, which is much smaller than that attained with 56 threads enabled (1.18). This result is different from that achieved by OCC, which suffers higher contention with more threads accessing the same records. This observation is still a consequence of the lemming effect, which results in pure serialized execution of database transactions, and therefore fewer transactions get aborted.

[1]The abort rate equals to the ratio the number of transaction re-executions to the number of committed transactions.

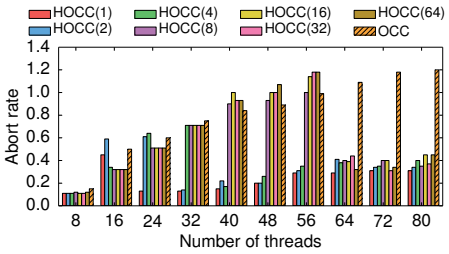


Figure 12: Database transaction abort rate of HOCC. The number of warehouses is set to 4.

To conclude, the state-of-the-art HTM-assisted OCC protocol may not be an attractive solution for supporting contended OLTP workloads.

Transaction rate. Next, we compare HTCC with several other concurrency control protocols. We set the restart threshold of all the HTM-assisted protocols to 64.

The following experiment evaluates the performance of each protocol under low-contention workloads. Figure 13 shows the result with the number of warehouses set to 40. All the four OCC-style protocols, including HTCC, OCC, SOCC, and HOCC, scale near-linearly when the thread count is increased from 1 to 40. With over 40 threads enabled, the performance stablizes, and stops improving. As explained earlier, this is because the hyper-threading scheme can produce sub-linear scalability, hence constraining the performance of the tested protocols. Thanks to the efficiency of HTM and work-set caching, HTCC yields over 10% higher transaction rate than OCC. Under the same scenario, the HTM-assisted HOCC only produces a similar performance with that of SOCC. Despite the absence of hardware transactional support, SOCC still generates a very competitive result, as it fully eliminates the necessity for tracking anti-dependency relations. Compared with these OCC-style protocols, 2PL achieves a lower transaction rate. The main reason is that it requires a longer locking duration for every record read or written by a transaction, and hence the overall performance is bounded. As a variation of timestamp ordering (TO) protocol, HTO still suffers high overhead for maintaining internal data structures [41], consequently leading to unsatisfactory results, even if HTM is utilized.

Next, we compare the performance of these protocols with the number of warehouses set to 4. In this scenario, the workload is highly contended. Figure 14 shows the transaction rate of each protocol with the thread count

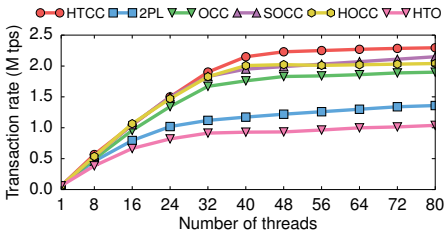


Figure 13: Performance comparison with the number of warehouses set to 40.

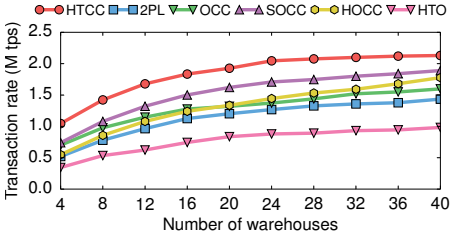


Figure 15: Performance comparison with the number of threads set to 40.

varied from 1 to 80. With 80 threads enabled, HTCC can process around 1,000 K transactions per second (tps), and this number is respectively 2 times and 6 times higher than that attained by SOCC and HOCC. OCC yields similar performance compared to 2PL, but the overhead for tracking anti-dependency makes it 10% slower than SOCC. The low performance of HOCC and HTO also indicate that the state-of-the-art HTM-assisted protocol cannot scale well under high-contention workloads.

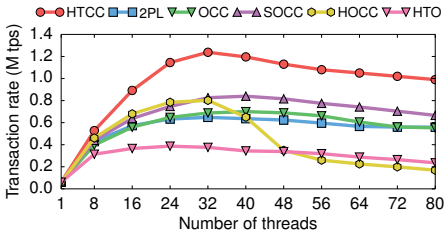


Figure 14: Performance comparison with the number of warehouses set to 4.

To further validate the scalability of HTCC, we measure the transaction rate achieved by each protocol with the number of warehouses changed from 4 to 40. Figure 15 shows the result with 40 threads enabled. Compared with all the other protocols, HTCC yields a superior transaction rate even under highly contended workloads. Specifically, the performance of HTCC is constantly 1.2 to 2 times higher than that achieved by SOCC. While HOCC produces good performance when the number of warehouses is set to 40, its unoptimized use of HTM makes it vulnerable to workload contentions.

All the experiment results reported above have confirmed that HTCC achieves much better performance than the existing (HTM-assisted) concurrency control protocols, especially under highly contended workloads.

Transaction latency. The following experiment measures the latency achieved by each OCC-style protocol. We omit the measurement for 2PL and HTO, as our previous experiments have already shown that these two protocols cannot attain high performance under various types of workloads. Figure 16 shows the corresponding results with 40 threads enabled. The number of warehouses is set to 4. Compared with the other three protocols, HTCC incurs a comparatively lower latency when processing NewOrder transactions. Specifically, 89% of the transactions can reach the final commitment within 160 µs. This number is much larger than that achieved by OCC (78%) and SOCC (73%). This is because HTCC attempts to repair any invalid accesses instead of restarting the entire transaction when an validation failure occurs. This approach essentially saves the resources that have been allocated for executing the transaction. In contrast to the other three protocols, HOCC gives rise to significantly higher transaction latency, and only 37% of the transactions can commit within 640 µs. The key reason is that the unoptimized use of HTM makes HOCC extremely vulnerable to conflicting accesses, and the lemming effect caused by frequent aborts eventually leads to pure serialized execution without parallelism, making the resultant transaction latency intolerable.

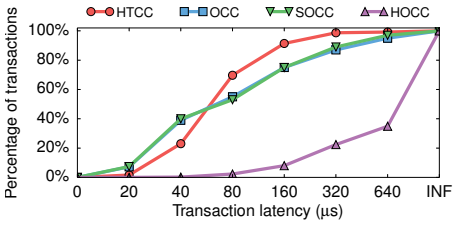


Figure 16: Comparison of transaction latency. The number of warehouses is set to 4.

Performance breakdown. HTCC attains robust transaction processing due to the use of a hybrid synchronization scheme, a workset-caching strategy, and a static-analysis-facilitated optimization. We next examine how these mechanisms contribute to HTCC's performance.

We first study the effectiveness of the hybrid synchronization mechanism. As shown in Table 1, when processing transactions on a database with 4 warehouses using OCC, 62.9% and 31.0% of the database transaction aborts are respectively caused by the conflicting accesses to the records in the WAREHOUSE and the DISTRICT

tables. Therefore, applying fine-grained locks on these records can potentially reduce the performance degradation caused by high abort rate. Keeping this observation in mind, we disable HTCC's automatic scheme for data classification and manually select different sets of hot records and measure the influence to HTCC's transaction rate. Figure 17 shows the result with the number of warehouses set to 4. When all the records are labeled as cold (i.e., HTM region is utilized to guard atomic accesses to the targeted data records), the performance of HTCC drops drastically once the thread count reaches 32 (see HTCC (None)). However, if all the records in the WAREHOUSE table are labeled as hot data, the performance degradation is remarkably mitigated (see HTCC (W)). In particular, HTCC attains a comparatively high performance with every record in the WAREHOUSE and the DISTRICT tables tagged as hot (see HTCC (W+D)). These results directly confirmed the effectiveness of fine-grained locking in protecting conflicting records. The same figure also reports HTCC's performance when all the records in the database are labeled as hot (see HTCC (All)). In this case, the effects of HTCC is indeed equivalent to that of 2PL. As shown in Figure 17, the resulting performance is still much lower than that of HTCC (W+D), and this essentially indicates the superior performance of HTM region when executing low-contention critical sections.

Table	4 WHs	10 WHs	40 WHs
WAREHOUSE	62.9%	56.6%	46.8%
DISTRICT	31.0%	37.2%	42.6%
Others	6.1%	6.2%	10.6%

Table 1: Data contentions in each table with 80 threads.

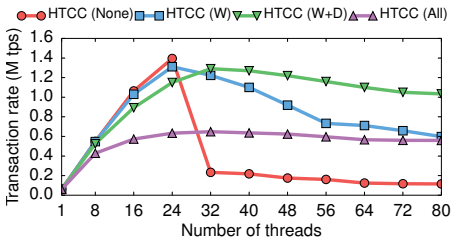


Figure 17: Effectiveness of hybrid concurrency control scheme. The number of warehouses is set to 4.

Figure 18 provides a performance analysis on how the workset caching strategy as well as the static-analysis-facilitated optimization benefit the transaction processing. The number of warehouses in this experiment is still set to 4, generating a highly contended workload. We measure the transaction rate produced by: (1) HTCC, (2) HTCC with static-analysis optimization disabled (see HTCC(-S)), (3) HTCC with both the workset caching and the static-analysis optimization disabled (see HTCC(-SC)), and (4) OCC. In this experiment, we have enabled hybrid processing scheme. Please note that HTCC(-SC) essentially tests the performance purely achieved by the hybrid processing scheme. As shown in Figure 18, the performance gaps among HTCC, HTCC(-S), HTCC(-SC), and OCC widen with the increase of thread count. When scaling to 40 threads, HTCC yields a transaction rate at around 1.2 M tps, which is respectively over 10% and 40% higher than that of HTCC(-S) and HTCC(-SC). This confirmed the effectiveness of both the workset caching and static-analysis-facilitated optimization in HTCC. The same experiment also shows that HTCC(-SC) achieves around 20% higher performance compared to OCC, and this further reaffirmed the efficiency of the hybrid concurrency control scheme in HTCC.

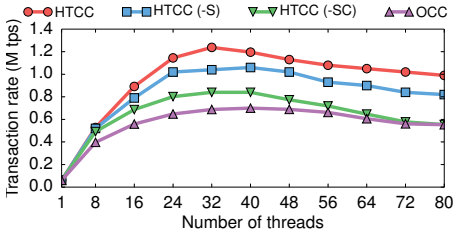


Figure 18: Effectiveness of workset caching and static-analysis-facilitated optimization. The number of warehouses is set to 4.

The delta-restoration mechanism adopted by HTCC incurs a certain amount of overhead to the system runtime. To quantify the overhead, we compare HTCC with HTCC(NV), a variation of HTCC that commits a transaction without invoking HTCC's validation phase. It should be noted that, due to the absence of the validation phase, HTCC(NV) will yield inconsistent query result. However, as it fully bypasses HTCC's overhead in restoring inconsistent operations, it essentially provides a higher bound on the performance that HTCC can potentially achieve. Figure 19 shows the performance comparison between HTCC and HTCC(NV). The number of threads is set to 40. As the number of warehouses increases, the performance of HTCC becomes closer to that of HTCC(NV). This is because increasing the number of warehouses will decrease the workload contention, and a transaction processed under lower contention is less likely to fail the validation phase. With the number of warehouses set to 4, HTCC(NV) can yield over 40% higher transaction rate, which reflects the overhead of operation restoration.

To sum up, the hybrid synchronization mechanism, the workset caching strategy, and the static-analysis-

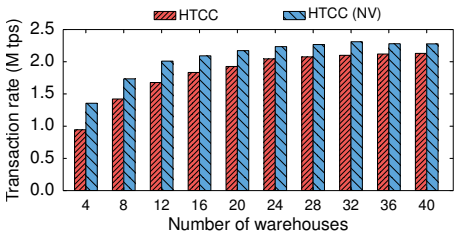


Figure 19: Overhead of delta-restoration. The number of threads is set to 40.

facilitated optimization scheme altogether provide a remarkable performance boost for HTCC.

7 RELATED WORK

Transaction processing. Extensive research efforts have been invested into the development of main-memory database systems. A pioneering work is Silo [32], a shared-everything database that attains high transaction rate using a variation of the conventional OCC protocol [15]. Researchers from Microsoft recently analyzed the performance of multi-version OCC [16, 18], and their study further lays the foundation for the Hekaton database [7]. However, neither Silo nor Hekaton is robust to contended workload, and Yu et al. [41]'s thorough scalability evaluation of traditional concurrency control protocols on 1000 cores further confirmed this point. Several recent works have proposed partitioning scheme to achieve higher scalability on multicore architectures [28]. Both DORA [25] and PLP [26], which are built on top of Shore-MT [12], partition data among cores to reduce long lock waiting time on a centralized lock manager. Several databases [13, 14, 31] employ a deterministic execution model that achieves high scalability using partition-level locks. While these databases can yield high transaction rate when the workload is well partitioned, the performance may severely degrade with an increasing number of distributed transactions. Leveraging program semantics can boost the performance of OLTP databases. Several works [24, 37, 42] leveraged transaction chopping [29] to speed up transaction processing and failure recovery. While HTCC optimizes its performance by using a similar static-analysis mechanism adopted in Wu et al. [36], its HTM-assisted hybrid protocol and caching mechanism can be directly applied to any ad-hoc transactions.

Transactional memory. Transaction memory is well studied in recent years [10, 11]. Sonmez et al. [30] proposed a mechanism that allows STM to dynamically select the best scheme for individual variables. While their design is similar with the hybrid protocol introduced in this paper, their work is restricted in the field of STM and did not consider the overhead caused by re-executing aborted transactions. Xiang et al. [39] also observed a high abort rate of HTM transactions and presented a consistency-oblivious (i.e., OCC-like) solution [3, 4] for reducing the HTM abort rate caused by capacity overflow, Their following work [38] further mitigated the conflict-caused abort problem using advisory lock. Litz et al. [20] borrowed the idea of snapshot isolation from the database community to reduce the abort rate. Different from these works, HTCC relies on a hybrid protocol and a lightweight caching mechanism to reduce the abort rate as well as the incurred abort overhead. Several recent works have exploited HTM to improve the performance of OLTP databases. Yoo et al. [40] utilized Intel's TSX to build efficient indexes, and Makreshanski et al. [21] further studied the interplay of HTM and lock-free indexing methods. Wang et al. [33] also employed HTM to build a concurrent skiplist. These studies on concurrent database indexes revealed that high abort rate due to capacity overflow and data contention can severely restrict HTM's performance. To deal with the high abort rate caused by HTM's capacity overflow, Leis et al. [17] and Wang et al. [34] respectively modified the timestamp ordering and OCC protocols to fully explore HTM's benefits in atomic execution. While achieving satisfactory performance when processing low-contention workloads, neither of them is able to sustain high transaction rate if the workload is contended. Wei et al. [35] and Chen et al. [6] exploited HTM and RDMA to build speedy distributed OLTP databases. As a departure from these works, HTCC focuses on exploiting the benefits of HTM for scalable and robust transaction processing on multicores.

8 CONCLUSION

We have proposed HTCC, an HTM-assisted concurrency control protocol that aims at providing scalable and robust in-memory transaction processing. HTCC attains its goal by reducing the transaction abort rate and minimizing the overhead of restarting aborted transactions. Our experiments confirmed that HTCC can yield high performance even under highly contended workloads. As a future work, we will explore how HTM can be leveraged to improve the performance of modern multi-version database systems.

Acknowledgment

We sincerely thank our shepherd Tim Harris and the anonymous reviewers for their insightful suggestions. This research is supported in part by a MOE/NUS grant R-252-000-586-112.

References

[1] http://llvm.org/docs/passes.html.

[2] http://www.tpc.org/tpcc/.

[3] AFEK, Y., AVNI, H., AND SHAVIT, N. Towards Consistency Oblivious Programming. In *OPODIS* (2011).

[4] AVNI, H., AND KUSZMAUL, B. C. Improving HTM Scaling with Consistency-Oblivious Programming. In *TRANSACT* (2014).

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems*. 1987.

[6] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and General Distributed Transactions using RDMA and HTM. In *EuroSys* (2016).

[7] DIACONU, C., FREEDMAN, C., ISMERT, E., LARSON, P.-A., MITTAL, P., STONECIPHER, R., VERMA, N., AND ZWILLING, M. Hekaton: SQL Server's Memory-Optimized OLTP Engine. In *SIGMOD* (2013).

[8] DICE, D., HERLIHY, M., LEA, D., LEV, Y., LUCHANGCO, V., MESARD, W., MOIR, M., MOORE, K., AND NUSSBAUM, D. Applications of the Adaptive Transactional Memory Test Platform. In *TRANSACT* (2008).

[9] ESWARAN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM 19*, 11 (1976).

[10] HARRIS, T., LARUS, J., AND RAJWAR, R. Transactional Memory. Morgan and Claypool Publishers.

[11] HARRIS, T., MARLOW, S., PEYTON-JONES, S., AND HERLIHY, M. Composable Memory Transactions. In *PPoPP* (2005).

[12] JOHNSON, R., PANDIS, I., HARDAVELLAS, N., AILAMAKI, A., AND FALSAFI, B. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *EDBT* (2009).

[13] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. In *VLDB* (2008).

[14] KEMPER, A., AND NEUMANN, T. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE* (2011).

[15] KUNG, H.-T., AND ROBINSON, J. T. On Optimistic Methods for Concurrency Control. *TODS 6*, 2 (1981).

[16] LARSON, P.-Å., BLANAS, S., DIACONU, C., FREEDMAN, C., PATEL, J. M., AND ZWILLING, M. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. In *VLDB* (2011).

[17] LEIS, V., KEMPER, A., AND NEUMANN, T. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *ICDE* (2014).

[18] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. Multi-Version Range Concurrency Control in Deuteronomy. In *VLDB* (2015).

[19] LI, X., ANDERSEN, D. G., KAMINSKY, M., AND FREEDMAN, M. J. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys* (2014).

[20] LITZ, H., CHERITON, D., FIROOZSHAHIAN, A., AZIZI, O., AND STEVENSON, J. P. SI-TM: Reducing Transactional Memory Abort Rates through Snapshot Isolation. In *ASPLOS* (2014).

[21] MAKRESHANSKI, D., LEVANDOSKI, J., AND STUTSMAN, R. To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-Free Indexing. In *VLDB* (2015).

[22] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *EuroSys* (2012).

[23] MOHAN, C. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *VLDB* (1990).

[24] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting More Concurrency from Distribted Transactions. In *OSDI* (2014).

[25] PANDIS, I., JOHNSON, R., HARDAVELLAS, N., AND AILAMAKI, A. Data-Oriented Transaction Execution. In *VLDB* (2010).

[26] PANDIS, I., TÖZÜN, P., JOHNSON, R., AND AILAMAKI, A. PLP: Page Latch-Free Shared-Everything OLTP. In *VLDB* (2011).

[27] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *POPL* (1995).

[28] SALOMIE, T.-I., SUBASU, I. E., GICEVA, J., AND ALONSO, G. Database Engines on Multicores, Why Parallelize When You Can Distribute? In *EuroSys* (2011).

[29] SHASHA, D., LLIRBAT, F., SIMON, E., AND VALDURIEZ, P. Transaction Chopping: Algorithms and Performance Studies. *TODS 20*, 3 (1995).

[30] SÖNMEZ, N., HARRIS, T., CRISTAL, A., ÜNSAL, O. S., AND VALERO, M. Taking the Heat Off Transactions: Dynamic Selection of Pessimistic Concurrency Control. In *IPDPS* (2009).

[31] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD* (2012).

[32] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy Transactions in Multicore In-Memory Databases. In *SOSP* (2013).

[33] WANG, Z., QIAN, H., CHEN, H., AND LI, J. Opportunities and Pitfalls of Multi-Core Scaling using Hardware Transaction Memory. In *APSys* (2013).

[34] WANG, Z., QIAN, H., LI, J., AND CHEN, H. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *EuroSys* (2014).

[35] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast In-memory Transaction Processing using RDMA and RTM. In *SOSP* (2015).

[36] WU, Y., CHAN, C.-Y., AND TAN, K.-L. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *SIGMOD* (2016).

[37] WU, Y., GUO, W., CHAN, C.-Y., AND TAN, K.-L. Parallel Database Recovery for Multicore Main-Memory Databases. In *CoRR* (2016).

[38] XIANG, L., AND SCOTT, M. L. Conflict Reduction in Hardware Transactions Using Advisory Locks. In *SPAA* (2015).

[39] XIANG, L., AND SCOTT, M. L. Software Partitioning of Hardware Transactions. In *PPoPP* (2015).

[40] YOO, R. M., HUGHES, C. J., LAI, K., AND RAJWAR, R. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In *SC* (2013).

[41] YU, X., BEZERRA, G., PAVLO, A., DEVADAS, S., AND STONEBRAKER, M. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. In *VLDB* (2014).

[42] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *SOSP* (2013).

Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality

Yue Cheng, *Virginia Tech*
Fred Douglis, *EMC*
Philip Shilane, *EMC*
Michael Trachtman, *EMC*
Grant Wallace, *EMC*
Peter Desnoyers, *Northeastern University*
Kai Li, *Princeton University*

Abstract

NAND-based solid-state (flash) drives are known for providing better performance than magnetic disk drives, but they have limits on endurance, the number of times data can be erased and overwritten. Furthermore, the unit of erasure can be many times larger than the basic unit of I/O; this leads to complexity with respect to consolidating live data and erasing obsolete data. When flash drives are used as a cache for a larger, disk-based storage system, the choice of a cache replacement algorithm can make a significant difference in both performance and endurance. While there are many cache replacement algorithms, their effectiveness is hard to judge due to the lack of a baseline against which to compare them: Belady's MIN, the usual offline best-case algorithm, considers read hit ratio but not endurance.

We explore offline algorithms for flash caching in terms of both hit ratio and flash lifespan. We design and implement a multi-stage heuristic by synthesizing several techniques that manage data at the granularity of a flash erasure unit (which we call a container) to approximate the offline optimal algorithm. We find that simple techniques contribute most of the available erasure savings. Our evaluation shows that the container-optimized offline heuristic is able to provide the same optimal read hit ratio as MIN with 67% fewer flash erasures. More fundamentally, our investigation provides a useful approximate baseline for evaluating any online algorithm, highlighting the importance of comparing new policies for caching compound blocks in flash.

1 Introduction

Unlike magnetic disk drives, flash devices such as solid state drives (SSDs) transfer data in one unit but explicitly *erase* data in a larger unit before rewriting. This erasure step is time-consuming (relative to transfer speeds) and it also has implications for the endurance of the device, as the number of erasures of a given location in flash is limited. A common endurance metric is Erasures Per Block Per Day (EPBPD), a rate commonly guaranteed by flash manufacturers for a time period [33, 34].

SSDs typically provide a flash translation layer (FTL) within the device, which maps from logical block numbers to physical locations. A host can access individual file blocks that are kilobytes in size, and if some live blocks are physically located in the same erasure unit as data that can be recycled, the FTL will garbage collect (GC) by copying the live data and then erasing the previous location to make it available for new writes.

As an alternative to performing GC in the FTL, the host can group file blocks to match the erasure unit (also called *blocks* in flash terminology). While some research literature refers to these groupings as "blocks" (e.g., RIPQ [38]), there are many other names for it: write-evict unit [22], write unit [31], erase group unit [30], and *container* in our own recent work on online flash cache replacement [23]. Thus we use "container" to describe these groupings henceforth.

Containers are written in bulk, thus the FTL never sees partially dead containers it needs to GC. However, the host must do its own GC to salvage important data from containers before reusing them. The argument behind moving the functionality from the SSD into the host is that the host has better overall knowledge and can use the SSD more efficiently than the FTL [21].

Flash storage can be used for various purposes, including running a standalone file system [7, 16, 20, 24, 41] and acting as a cache for a larger disk-based file system [8, 9, 13, 19, 29, 32, 36, 42]. The latter is a somewhat more challenging problem than running a file system, because a cache has an additional degree of freedom: data can be stored in the cache or bypassed [15, 35], but a file system must store all data. In addition, the cache may have different goals for the flash storage: maximize hit rate, regardless of the effect on flash endurance; limit flash wear-out, and maximize the hit rate subject to that limit; or optimize for some other utility function that takes both performance and endurance into account [23].

Flash cache replacement solutions such as RIPQ [38] and Pannier [23] consider practical approaches given a historical sequence of file operations; i.e., they are "online" algorithms. Traditionally, researchers compare online algorithms against an offline optimal "best case" baseline to assess how much room an algorithm might have for improvement [5, 9, 11, 43]. For cache replacement, Belady's MIN algorithm [2] has long been used as that baseline.

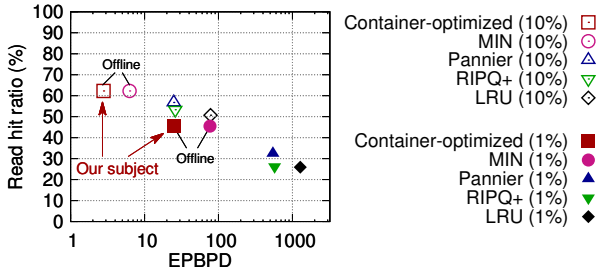


Figure 1: A RHR vs. endurance (EPBPD, on a log scale) scatter-plot of results for different caching algorithms. We report the average RHR and EPBPD across a set of traces, using cache sizes of 1% or 10% of the WSS. (Descriptions of the datasets and the full box-and-whisker plots appear below.) RIPQ+ and Pannier are online algorithms, described in §5.3. The goal of our container-optimized offline heuristic is to reduce EPBPD with the same RHR as MIN, an offline algorithm that provides the optimal RHR without considering erasures.

Figure 1 plots read hit ratio (RHR) against the EPBPD required by a given algorithm assuming SSD sizes of 1% or 10% of the working set size (WSS) of a collection of traces (described in §5.1). MIN achieves an average RHR improvement of 10%–75% compared to LRU, a widely used online algorithm. Using future knowledge makes a huge difference to RHR: Pannier covers only 33% to 52% of the gap from LRU to MIN, while RIPQ+ (described in §5.3) sees about the same RHR as LRU. However, MIN is suboptimal when considering not only RHRs but also flash endurance: it will insert data into a cache if it will be accessed before something currently in the cache, even if the new data will itself be evicted before being accessed. Such an insertion increases the number of writes, and therefore the number of erasures, without improving RHR; we refer to these as *wasted writes*. In this paper we explore the tradeoffs between RHR and erasures when future accesses are known.

A second complicating factor arises in the context of flash storage: containers. Not only should an offline algorithm not insert unless it improves the RHR, it must be aware of the container layout. For example, data that will be invalidated around the same time would benefit by being placed in the same container, so the container can be erased without the need to copy blocks.

We believe that an offline *optimal* flash replacement algorithm not only requires future knowledge but is computationally infeasible. We develop a series of heuristics that use future knowledge to make best-effort cache replacement decisions, and we compare these heuristics to each other and to online algorithms. Our *container-optimized offline heuristic* maintains the same RHR as MIN. The heuristic identifies a block that is inserted into the cache but evicted before being read, and omits that insertion in order to avoid needing to erase the region where that block is stored. At the same time, the heuristic consolidates blocks that will be evicted at around the same time into the same container when possible, to minimize GC activities. One important finding is that simple techniques (e.g., omitting insertions and GC copies that are never reread) provide most of the benefit; the more complicated ones (e.g., consolidating blocks that die together) have a marginal impact on saving erasures at relatively large cache sizes. Alternatively, we describe other approaches to maximize RHR subject to erasure constraints. Figure 1 provides examples in which the average RHR is the same maximum achievable by MIN, while lowering EPBPD by 56%–67%. Interestingly, the container-based online algorithms reduce EPBPD relative to LRU by similar factors.

Specifically, we make the following contributions:

- We thoroughly investigate the problem space of offline compound object caching in flash.
- We identify and evaluate a set of techniques to make offline flash caching container-optimized.
- We present a multi-stage heuristic that approximates the offline optimal algorithm; our heuristic can serve as a useful approximate baseline for analyzing any online flash caching algorithm.
- We experiment with our heuristic on a wide range of traces collected from production/deployed systems, to validate that it can provide a practical upper bound for both RHR and lifespan.

2 Background and Related Work

Here we provide a brief background of the offline caching algorithms, discuss the challenges of finding an offline optimal caching algorithm for container-based flash, and describe some previous analytical efforts.

2.1 Belady's MIN and its Limitations

Belady's MIN algorithm [2] replaces elements whose next reference is the furthest in the future, and it is provably optimal with respect to the read hit ratio given certain assumptions [25, 26]. In particular, it applies in a single level of a caching hierarchy in which all blocks (or pages) *must* be inserted. For instance, it applies to demand-paging in a virtual memory environment.

Our environment is slightly different. We assume a DRAM cache at the highest level of the cache hierarchy and a flash device serving as an intermediate cache between DRAM and magnetic disks. A block that is read from disk into DRAM and then evicted from DRAM can be inserted into flash to make subsequent accesses faster, but it can also be removed from DRAM without inserting into flash ("read-around"). Similarly, writes need not be inserted into flash as long as persistent writes will be stored on disk (see §3.4).

Since this is not a demand-fetch algorithm, MIN is not

necessarily the optimal strategy. Consider a simple 2-location cache with the following access sequence:

A,B,C, A,B,D, A,B,C, A,B,D ...

In a demand-fetch algorithm a missing block must be inserted into the cache, replacing another one; in this case the hit rate will be $\frac{1}{3}$, as B will always be replaced by C or D before the next access. With read-around it is not necessary for C and D to be inserted into cache, allowing hits on both A and B for a hit rate of $\frac{2}{3}$. We note, however, that such behavior may be emulated by a demand-fetch algorithm using one more cache location, which is reserved for those elements which would not be inserted into cache in the read-around algorithm. The hit rate for a read-around algorithm with N cache locations is thus bounded by the performance of MIN with N+1 locations, a negligible difference for larger values of N which we ignore in the remainder of the paper.

Even if MIN provides the optimal RHR, we argue below that it can write more blocks than another approach providing the same RHR with fewer erasures. We use MIN to refer to a variant of Belady's algorithm that does not insert a block into the cache if it will not be reread, while M^+ is a further enhancement that does not insert a block that will not be reread *prior to eviction*.

Temam [39] extends Belady's algorithm by exploiting spatial locality to take better advantage of processor caches. Gill [10] applies Belady's policy to multi-level cache hierarchies. His technique is useful for iterating across multiple runs of a cache policy. However, since Belady targets general local memory caching, it is not directly applicable to container-based flash caching due to the inherent difference between DRAM and flash.

2.2 Container-based Caching Algorithms

Previous work shows that various container-based flash cache designs lead to different performance–lifespan trade-offs [22, 23, 30, 31, 38]. SDF [31], Nitro [22], and SRC [30] use a large write unit aligned to the flash erasure unit size to improve cache performance. RIPQ [38] leverages another level of indirection to track reaccessed photos within containers. Pannier [23] explicitly exploits hot/cold block and invalidation mixtures for container-based caching to further improve performance and reduce flash erasures. However, it is not known how much headroom in both performance and lifespan might exist for any state-of-the-art flash caching algorithms. To give a clear idea of how well an online flash caching algorithm performs, we need an offline optimal algorithm that incorporates performance and lifespan of the flash cache.

2.3 Analytical Approaches

Considerable prior work has explored the offline optimality of caching problems in various contexts from a theoretical perspective. Albers et al. [1] and Brehob et al. [4] prove the NP-hardness of optimal replacement for non-standard caches. Chrobak et al. [6] prove the strong NP-completeness of offline caching supporting elements with varying sizes (i.e., costs). Neither explicitly studies the offline optimality of the flash caching problem with two goals that are essentially in conflict.

Other researchers have looked at related problems. Horwitz et al. [14] formulate the index register allocation problem to the shortest path problem with a general graph model and prove the optimality of the allocation algorithm. Ben-Aroya and Toledo [3] analyze a variety of offline/online wear-leveling algorithms for flash-based storage systems. Although not directly related to our problem, these works provide insights into the offline optimality of container-based flash caching.

3 Quest for the Offline Optimal

A flash device contains a number of *flash blocks*, the unit of erasures, referred to in our paper as *containers* to avoid confusion with file blocks. But many of the issues surrounding flash caching arise even in the absence of containers. We refer to the case where each file block can be erased individually as **unit caching**, and we describe the metrics (§3.1) and algorithms (§3.2) in that context. This separates the general problem of deciding what to write into the flash cache in the first place from the overhead of garbage collecting containers; we return to the impact of containers in §3.3. In §3.2–3.3 we also introduce a set of techniques for eliminating wasted writes. We then discuss how to handle user-level writes in §3.4. Finally, we summarize the algorithms of interest in §3.5.

3.1 Metrics

The principal metrics of concern are:

Read Hit Ratio (RHR): The ratio of read I/O requests satisfied by the cache (DRAM cache + flash cache) over total read requests.

The Number of Flash Erasures: In order to compare the impact on lifespan across different algorithms and workloads, we focus on the EPBPD required to run a given algorithm on a given workload and cache size. The total number of erasures is the product of EPBPD, capacity, and workload duration.

Flash Usage Effectiveness (FUE): The FUE metric [23] endeavors to balance RHR and erasures. It is defined as the number of bytes of flash hit reads divided by flash writes, including client writes and internal copy-forward (CF) writes. A score of 1 means that, on average, every byte written to flash is read once, so higher scores are better. It can serve as a utility function to evaluate different algorithms. We define *Weighted FUE* (**WFUE**), a variant of FUE that considers both RHR and erasures and uses a weight to specify their relative importance:

Technique	Description	C
R_N	omit insertions reread $< N$ times	✗
TRIM	notify GC to omit dead blocks	✓
CFR	avoid wasted CF blocks	✓
E	segregate blocks by evict time	✓

Table 1: Summary of offline heuristic techniques used for eliminating wasted writes to the flash cache. C: container-optimized.

$$\text{WFUE} = \alpha * (\text{RHR}_A/\text{RHR}_M) + (1-\alpha) * (E_M - E_A)/E_M$$

The utility of an algorithm is determined by comparing the RHR and erasures (E) incurred by the algorithm, denoted by A, to the values for M^+ (an improved MIN, described in §3.2, denoted here by M for simplicity).[1] If α is low and an algorithm saves many writes in exchange for a small reduction in RHR, WFUE will increase.

3.2 Objectives and Algorithms

Depending on the goals of end users, we may have different objective functions. Optimizing for RHR irrespective of erasures is trivial: the performance metric RHR serves as a naïve but straightforward goal for which MIN can easily get an optimal solution, without considering the flash endurance. Taking erasures into account, we identify three objectives of interest. We describe each briefly to set the context for comparison, then elaborate on heuristics to optimize for them. (We do not claim their optimality, leaving such analysis to future work.)

O1: Maximal RHR The purpose of objective **O1** is to *minimize erasures subject to maximal RHR*. If we consider the RHR obtained by MIN, there should be a sequence of cache operations that will preserve MIN's hit ratio while reducing the number of erasures. Belady's **MIN** caches any block that either fits in the cache, or which will be reaccessed sooner than some other block in the cache. It does not take into account whether the block it inserts will itself be evicted from the cache before it is accessed.

The first step to reducing erasures while keeping the maximal RHR is to identify *wasted cache writes* due to eviction. **Algorithm M+** is a variant of MIN that identifies which blocks are added (via reads or writes) to the cache and subsequently evicted without rereference, then no longer inserts them into the cache (R_N in Table 1, where $N = 1$).

It is unintuitive, but cache writes can be wasted even if they result in a read hit. As an example, assume block A is in cache at time t_0, and will next be accessed at time $T > t_0$. If block B is accessed at t_0, and will be accessed exactly one more time at time $T-1$, MIN dictates that A be replaced by B. However, by removing B, there is still one miss (on B rather than A), while an extra write has occurred. Leaving A in the cache would have the same RHR but one fewer write into the cache.

Ultimately, our goal is to identify a Pareto-optimal solution set where it is impossible to reduce the number of erasures without reducing RHR. This requires that no block be inserted if it does not improve RHR, but the complexity of considering every insertion decision in that light is daunting. Thus we start with eliminating cache insertions that are completely wasted and leave additional trades of one miss against another to future work.

An offline heuristic **Algorithm H** that approximates **M+** works as follows:

Step 1 Annotate each entry with its next reference.

Step 2 Run **MIN** to annotate the trace with a sequence of cache insertions and evictions, given a cache capacity. Note all insertions that result in being evicted without at least one successful reference.

Step 3 Replay the annotated trace: do not cache a block that had not been accessed before eviction.

O2: Limited Erasures In some cases a user will be willing to sacrifice RHR in order to reduce the number of erasures. In fact, given limits on the total number of erasures of a given region of flash, it may be essential to make that tradeoff. Thus, **O2** first limits erasures to a particular rate, such as 5 EPBPD. (The EPBPD rate is multiplied by the size of the flash cache and the duration of the original trace to compute a total budget for erasures.) *Given an erasure limit, the goal of **O2** is to maximize RHR.* Note that the rate of erasures is averaged across an entire workload, meaning that the real limit is the total number of erasures; EPBPD is a way to normalize that count across workloads or configurations.

We can modify **Algorithm H** for **O2** to have a threshold. $\mathbf{H}_T$ works as follows:

Step 1 Run **H** and record all insertions. Annotate each record with the number of read hits absorbed as a result of that insertion, and count the total number of insertions resulting in a given number of read hits.

Step 2 Compute the number of cache insertions I performed in the run of **H** and the number of insertions I' allowed to achieve the EPBPD threshold T. If $I > I'$ then count the cumulative insertions CI resulting in 1 read hit, 2 read hits, and so on until $I - CI = I'$. Identify the reuse count, R, at which eliminating these low-reuse insertions brings the total EPBPD to the threshold T. Call the number of cache insertions with R reuses that must also be eliminated the leftover, L.

Step 3 Rerun **H**, skipping all insertions resulting in

[1] Though these two factors have different ranges and respond differently to changes, WFUE controls the value of both via normalization so that the higher each factor yields, the better the algorithm performs with respect to that goal. A negative value due to high erasures would demonstrate the deficiency of the algorithm in saving erasures. Hence, WFUE can serve as a general metric for quantitatively comparing different heuristics.

fewer than R read hits, and skipping the first L insertions resulting in exactly R hits.

Algebraically, we can view the above description as follows: Let A_i represent the count of cache insertions absorbing i hits.

$$I = \sum_{i=1}^{n} A_i \qquad \text{and} \qquad CI = (\sum_{i=1}^{R-1} A_i) + L$$

We identify R such that this results in $I - CI = I'$.

O3: Maximize WFUE The goal of **O3** is to *maximize WFUE*, which combines RHR and erasures into a single score to simplify the comparison of techniques (§3.1). Intuitively, the user may want to get the highest read hits per erasure (i.e., best "bang-for-the-buck" considering the user pays the cost of device endurance for RHR).

To compare the tradeoffs between RHR and erasures, we consider a variant of **H**, **Algorithm $\mathbf{H}_N$**, which omits cache insertions that are reread $< N$ times (R_N in Table 1). This is similar to the threshold-based **Algorithm $\mathbf{H}_T$**, but the decision about the number of reaccesses necessary to justify a cache insertion is *static*. An increase in the minimum number of reads per cache insertion should translate directly to a higher FUE, though the writes due to GC are also a factor. For WFUE, the value of α determines whether such a threshold is beneficial.

3.3 Impact from Containers

The metrics and objectives described in §3.1–3.2 apply to the unit caching scenario, in which each block may be erased separately, but they also apply to the container environment. The aim is still to minimize erasures subject to a maximal RHR, to maximize RHR subject to a limit on erasures, or to maximize a utility function of the two.

However, the approach to *solving* the optimization problem varies when containers are considered. This complexity arises because there is an extra degree of freedom: not only does an algorithm need to decide *whether* to cache a block, it must decide *where* it goes and whether to reinsert it during GC. Regarding placement, one option is to cache data in a container-oblivious manner. For instance, a host could write each block to a distinct location in flash and rely on an FTL to reorganize data to consolidate live data and reuse areas of dead data. This might result in a significant overhead from the FTL unwittingly copying forward blocks that MIN knows are no longer needed, so adding the SSL *TRIM* [40] command to inform the FTL that a block is dead can reduce erasures significantly (see §6.1). As shown in Table 1, we categorize TRIM as *container-optimized*, because an FTL itself manages data at the granularity of containers.

For CF, the first step is to supplement the annotations from §3.2 with information about blocks that are CF and not reaccessed. Copy-Forward Reduction (*CFR* in Table 1) effectively extends TRIM with the logic of R_1, by identifying "wasted" CFs; however, eliminating *all* needless CFs is difficult. With the smallest cache, on average this reduces the erasures due to wasted CFs from 4% to 1%; repeating this step a few more times brings it down another order of magnitude but does not completely eliminate wasted CFs. This is because (for a small cache) there is always a block to CF that has not yet been verified as a useful copy nor marked as wasted. Note that while it seems appealing to simply not copy something forward that was not copied forward in a previous iteration, the act of excluding a wasted copy makes new locations for data available, perturbing the sequence of operations that follow. This makes the record from the previous run only partly useful for avoiding mistakes in the subsequent run, an example of the "butterfly effect."

Still, writing in a container-optimized manner can improve on the naive data placement of M^+, which uses the FTL to fill a container at a time. By segregating blocks by their expiration time as they are written to flash (E in Table 1), we may be able to erase one container, without the need to CF, as soon as all the blocks within it are no longer needed. We refer to E as *container-optimized* since it explicitly consolidates data that die together at the host or application level.

Since the purpose of our study is to provide a best-case comparison point for real-world cache replacement algorithms, we focus henceforth on the container-based cache replacement policies. Note that if containers consist of only one block, any approaches that are specifically geared to containers should work for the unit caching replacement policy. In the next section, we describe the algorithms in greater detail, using **C** to represent the offline heuristic **H** in the context of containers.

3.4 Impact from Dirty Data

The results from the various algorithms depend significantly on how the cache treats writes into the file system. For example, Pannier [23] requires that all file writes be inserted into the cache, with the expectation that the cache be an internally consistent representation of the state of the file system at any given time. All writes are immediately reflected in stable storage, which is appropriate for an online algorithm; Pannier's comparison to Belady used the same approach even with future knowledge, writing all user-level writes into SSD.

With future knowledge, however, one can argue that a "dead write" that will be overwritten before being read need not be inserted into the cache. The same is true of a write that is never again referenced, though in that case it should be written through to disk. Since we model only the flash cache RHR and endurance, we place these dead writes into DRAM but not into flash.

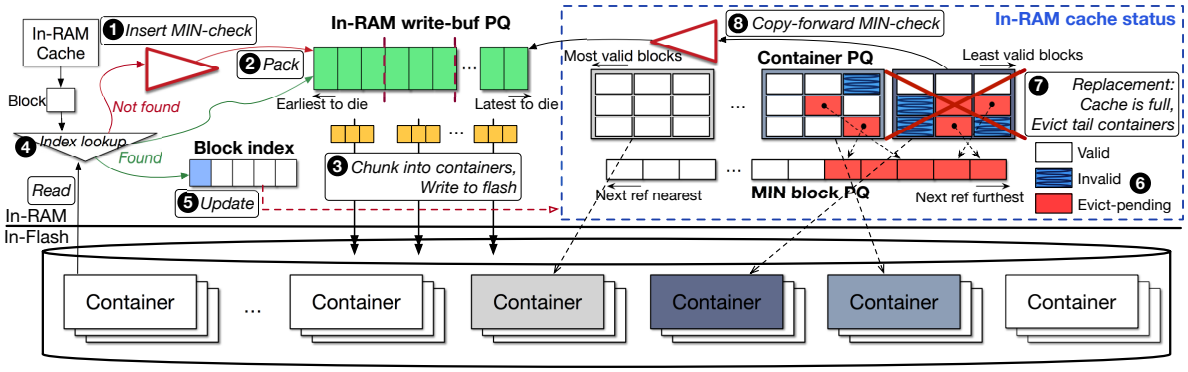


Figure 2: Container-optimized offline flash caching framework. PQ: priority queue.

3.5 Algorithm Granularity

For the remainder of this paper, we compare the following algorithms. **M** refers to variants of MIN while **C** refers to container-optimized algorithms. A table summarizing these (and other) algorithms appears in §5.3.

M Belady's MIN, which does not insert a block that will be overwritten or never reread by the client.

M+ A variant of MIN, which identifies a block that is inserted into cache but evicted before being read, and omits that insertion. It also uses TRIM to avoid CF once the last access to a block occurs.

$\mathbf{M}_N$ A variant of M^+, which does not insert blocks with accesses $< N$. M_1 is equivalent to M^+, while M_N generalizes it to $N > 1$.

$\mathbf{M}_T$ A variant of M^+, which eliminates enough low-reuse cache insertions to get the best RHR under a specific erasure limit (see §3.2).

C Each block is inserted if and only if M^+ would insert it. However, a write buffer of W containers is set aside in memory, and the system GCs whenever it is necessary to make that number of containers free to overwrite. Containers are filled using the *eviction timestamp* indicating when, using M^+, a block would be removed from the cache. The contents of the containers are ordered by eviction time, so the first container has the m blocks that will be evicted first, the next container has the next m blocks, and so on.

$\mathbf{C}_N$, $\mathbf{C}_T$ Analogous to M_N, M_T.

4 Offline Approximation

Here we describe how to evolve the container-oblivious MIN algorithm to a container-optimized heuristic C, which provides the same RHR but significantly fewer erasures. We also explain creating C_N and C_T.

4.1 Key Components

Figure 2 depicts the framework architecture and examples of the insertion and lookup paths. A detailed discussion appears in the next subsection, but the following components are the major building blocks.

Block Index An in-memory index maps from a block's key (e.g., LBA) to a location in flash. Upon a read, the in-memory index is checked for the block's key, and if found, the flash location is returned. Newly inserted blocks are added to the in-memory write buffer queue first. Once the content of the writer buffer is persisted in the flash cache, all blocks are assigned an address (an index entry) used for reference from the index. When invalidating a block, the block's index entry is removed.

In-RAM Write Buffer An in-memory write buffer is used to hold newly inserted blocks and supports in-place updates. The write buffer is implemented as a priority queue where the blocks are ranked based on their eviction timestamps (described in greater detail in §4.2). The write buffer queue is filled cumulatively and updated in an incremental fashion. Once the write buffer is full, its blocks are copied into containers, *sealed* and *persisted* in the flash cache. The advantage of cumulative packing and batch flushing is that the blocks with close eviction timestamps get allocated to the same container so that erasures are minimized. Overwrites to existing blocks stored in flash are redirected to the write buffer without updating the sealed container.

In-RAM Cache Status A few major in-memory data structures construct and reflect the runtime cache status. Once a container is sealed, its information structure is inserted into a **container priority queue** (PQ), a structure to support container-level insertion and eviction. Whenever a container is updated (e.g., a block is invalidated or evict-pending, etc.), its relevant position in the queue is updated. In addition to the container PQ, a **block-level MIN PQ** is designed to support the extended Belady logic and track the fine-grained block-level information. We discuss the operations of the MIN PQ in §4.2.

4.2 Container-optimized Offline Heuristic

The multi-stage heuristic C offers the optimal RHR while attempting to approach the practically lowest number of erasures on the flash cache. In the following, we describe the container-optimized heuristic pipeline (Figure 2) in

```
1 void Lookup(Object obj):
2   // WB: Write buffer priority queue
3   if WB.exist(obj.key) or INDEX.exist(obj.key):
4     Object existing = Read(obj.key)
5     OnAccess(existing, obj)
6   else: OnInsert(obj) // Upon a miss
7
8 void OnInsert(Object obj):
9   if not MINFull():
10    if obj.next_ref == INF: return
11    if Rec[obj.key].read_freq <= read_freq_thresh:
12      return
13    MIN_Q.insert(obj) // Insert into MIN queue
14  else:
15    Object victim = MIN_Q.top()
16    if obj.next_ref > victim.next_ref:  return
17    if Rec[obj.key].read_freq <= read_freq_thresh:
18      return
19    // Trigger evict on MIN queue
20    EvictMIN(MIN_Q, victim)
21    MIN_Q.insert(obj)
22  if WB.full(): OnSeal()
23  WB.insert(obj) // Pack into WB
24  EvictFlash()
25
26 void OnSeal():
27  Object obj = WB.begin()
28  // Iterate through all sorted objs
29  while obj != WB.end():
30    FreeCList[curr_ptr].insert(obj)
31    if FreeCList[curr_ptr].full():
32      // C_Q: Container queue
33      C_Q.insert(FreeCList[curr_ptr++])
34    obj = WB.next()
35  WB.clear()
36
37 void EvictMIN(Object victim):
38  MIN_Q.pop();
39  if WB.exist(victim.key): // Remove if in WB
40    WB.erase(victim)
41    return
42  Container c = GetContainer(victim)
43  victim.evict_pending = true
44  c.num_evict_pending++
45  C_Q.update(c) // Update c's position in C_Q
46
47 void EvictFlash():
48  while FlashFull():
49    Container c = C_Q.pop()
50    GC(c) // Garbage collect the evicted c
51
52 void OnCopyForward(Object obj):
53  if obj.next_ref == INF: return
54  MIN_Q.insert(obj)
55
56 void OnAccess(Object old_obj, Object new_obj):
57  if old_obj.evict_pending:
58    Count access as a miss
59    return
60  Update hit stats
61  old_obj.next_ref = new_obj.next_ref
62  // Update obj's position in MIN queue
63  MIN_Q.update(old_obj)
64  old_obj.evict_time = new_obj.evict_time
```

Figure 3: Functions handling events for flash-cached blocks and containers in the container-optimized offline heuristic.

detail. Figure 3 shows the pseudocode of how the offline heuristic handles different events.

Insert, Access and Seal We describe inserting a block, accessing a block, and sealing/persisting a container. ❶ When a client inserts a block upon a miss and the cache is not full, the `OnInsert` function first checks if the block's next reference timestamp is `INFINITE` (i.e., the block is never read again in the future or the next reference is a write). If so, `OnInsert` simply bypasses it and returns. Otherwise, an insertion record is checked to see if the block exceeds `read_freq_thresh`, a configurable read hit threshold. For instance, setting the `read_freq_thresh` to 1 filters out those that are inserted but evicted before being read, avoiding a number of

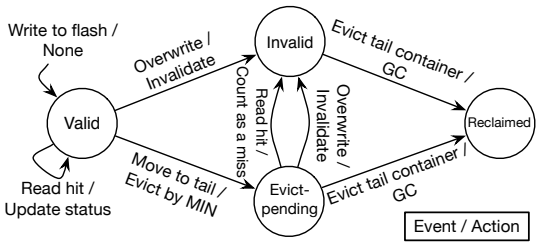


Figure 4: State transitions of blocks in our heuristic.

wasted writes. C_N and C_T also take advantage of this scheme for trading-off RHR with endurance: setting the threshold higher additionally filters out the less *useful* writes, reducing RHR but decreasing the number of erasures. The threshold can be fixed (C_N) or computed based on the total erasure budget (C_T, as described for H_T in §3.2). More useful writes, which result in a greater number of read hits, still take place. We study the trade-offs in §6.3.

Once the threshold test succeeds, if the cache is full and the block will be referenced furthest in the future (i.e., the block has a greater next reference timestamp than the most distant block (`victim`) currently stored in the cache), `OnInsert` returns without inserting it. When both checks are passed, `EvictMIN` function is triggered to evict the `victim` and the new block is inserted into the MIN queue (`MIN_Q`). At the same time, ❷ the block is added to the in-memory write buffer queue (`WB`).

❸ When `WB` is full, all the blocks held in it, sorted based on their eviction timestamp, are copied into multiple containers from the free container list `FreeCList`. `curr_ptr` maintains a pointer to the first available free container in `FreeCList`. (We compare the sorted approach to FIFO insertion of the blocks in §6.2.) The `OnSeal` function then persists the open containers in the flash cache.

Lookup ❹ On a `Lookup`, both the `WB` and in-memory index (`INDEX`) are referenced to locate the block, and the read is serviced. On a read access (`OnAccess`), ❺ the read hit updates the existing block's block-level metadata (`old_obj`) and `old_obj`'s position is updated in `MIN_Q` on the next access time. Upon a miss `OnInsert` is triggered as described.

Invalidation and Eviction ❻ The container-optimized offline caching introduces another new block state – *evict-pending*. Evict-pending describes the state when a block is evicted from `MIN_Q` (transitioning from the *valid* state to evict-pending) but temporarily resides in the GC area of the flash, pending being reclaimed. Figure 4 shows the state transitions of a block in the heuristic. A block is inserted/reinserted into the flash cache with a valid state. Once it is overwritten, the old block in the flash is marked as invalid and the updated data is inserted into `WB`. Overwriting an evict-pending block makes it transition to the invalid state. If the `victim` to be evicted

from MIN_Q happens to reside in WB, EvictMIN directly removes it from the memory. The on-flash container maintains a num_evict_pending counter. On evict-pending, the corresponding container increments its counter and updates its position in the container PQ C_Q.

Let V, I, and E represent the percentage of valid, invalidated and evict-pending blocks in a container, respectively; then $V+I+E=100\%$. The priority of a container is calculated using V. When the cache is full, EvictFlash selects the container with the lowest V (i.e., the fewest valid blocks) for eviction.

Copy-forwarding and GC ❼ When the cache is full and a container has been selected for eviction, the heuristic copies valid blocks forward to the in-memory write buffer. Function OnCopyForward is called to check if the reinserted block is useful. All the invalidated and evict-pending blocks get erased in the flash. The selected container is then reclaimed and inserted back to FreeCList.

❽ The check for a "useful" reinserted block looks for future references and (optionally) confirms the block will not be evicted before it is read.

5 Experimental Methodology

Throughout our analyses, we set the flash erasure unit size to 2MB. We place a small DRAM cache (5% of the flash cache size) in front of the flash cache to represent the use case where the flash cache is used as a second-level cache. (The DRAM cache uses the MIN eviction policy for offline flash cache algorithms and LRU for online ones.) Because some of the datasets in the repositories we accessed have too small a working set for 5% of the smallest flash cache to hold the maximum number of in-memory containers, we restrict our analyses to those datasets with a minimum 32GB working set.

This section describes the traces; implementation and configuration for the experimental system; and the set of caching algorithms evaluated.

5.1 Trace Description

We use a set of 34 traces from 3 repositories:

EMC-VMAX Traces: This includes 25 traces of EMC VMAX primary storage servers [37] that span at least 24 hours, have at least 1GB of both reads and writes, and meet the minimum working set threshold (slightly over half the available traces).

MS Production Server Traces: This includes 3 storage traces from a diverse set of Microsoft Corporation production servers captured using event tracing for windows instrumentation [18], meeting the 32GB minimum.[2]

MSR-Cambridge Traces: This includes 6 block-level traces lasting for 168 hours on 13 servers, representing a typical enterprise datacenter [28]. We narrowed available traces to 6 to include appropriate traces for cache studies. The properties include a working set size greater than 32GB, $\geq 5\%$ of capacity accessed, and read/write balance ($\leq 45\%$ writes).[3]

Given a raw trace, we annotate it by making a full pass over the trace and marking each 4KB I/O block with its next reference timestamp. Large requests in traces are split into 4KB blocks, each of which is annotated with the timestamp of its next occurrence individually. The annotated trace is then fed to the cache simulator. Round 1 simulation, e.g., M, may generate the insert log that can be used by round 2 simulation (e.g., M^+, M_N) to filter out blocks that will be evicted before being read.

5.2 Implementation and Configuration

For the experimental evaluation, we built our container-optimized offline caching heuristic by adding about 3,900 lines of C++ code to a full-system cache simulator. It reports metrics such as hit ratio and flash erasures based on the Micron MLC flash specification [27].

The size of the flash cache for each trace is determined by a fixed fraction of the WSS of the trace (from 1–10%). For C, a write buffer queue (with default size equal to 4 containers) is used for newly inserted blocks and is a subset of this DRAM cache. We over-provision the flash capacity by 7% by default; this extra capacity is used for FTL GC or to ensure the ability to manually clean containers in the container-optimized case. We discuss varying the over-provisioning space in §6.2.

We conduct the simulation study on 4 VMs each equipped with 4 cores and 128 GB DRAM. All tests are run in parallel (using GNU parallel [12]). We measured the CPU time of heuristic M^+ and C looping over all traces, not including the runtime of trace annotating and insertion log generating pre-runs. We ran the experiments 5 times and variance was low. C takes 21.7% longer (2.47 hr) than M^+ (2.03 hr) for the smallest cache size due to the overhead of PQ used by C under intensive GCs. The heuristic keeps track of more metadata in 10% cache size. Thus, with the least amount of GCs, it takes M^+ almost as long (2.19 hrs) as C does (2.21 hrs), to replay all traces. The results show that our heuristic simulation can process a large set of real-world traces within a reasonable time. This strengthens our confidence that our offline heuristics can serve as a practically useful tool.

5.3 Caching Algorithms

Table 2 shows the caching algorithms selected to represent past and present work. We select the classic LRU algorithm, two state-of-the-art container-based online algorithms (described next), and a variety of offline algo-

[2] The traces are: BuildServer, DisplayAdsPayload, DevelopmentToolsRelease.

[3] The traces are: prn0, prn1, proj0, proj4, src12, usr2.

Policy	Abbrev.	Description	O	C
LRU	L	least recently used	✗	✗
RIPQ+	R^+	RIPQ with overwrites, segmented-LRU	✗	✗
Pannier	P	container-based, S2LRU*, survival queue, insertion throttling	✗	✓
MIN	M	FK, do not insert data whose next ref. is the most distant	✓	✗
MIN+	M^+	FK, do not insert data evicted without read (R_1+TRIM, **O1**)	✓	✓
MIN+write-threshold	M_T	FK, limit number of insertions (R_T+TRIM, **O2**)	✓	✓
MIN+insertion-removal	M_N	FK, do not insert data with accesses $< N$ (R_N+TRIM, **O3**)	✓	✓
Container-optimized	C	FK, container-optimized (R_1+TRIM+CFR+E, **O1**)	✓	✓
C+write-threshold	C_T	FK, container-optimized (R_T+TRIM+CFR+E, **O2**)	✓	✓
C+insertion-removal	C_N	FK, container-optimized, do not insert data w/ acc. $< N$ (R_N+TRIM+CFR+E, **O3**)	✓	✓

Table 2: Caching algorithms. FK: future knowledge, O: offline, C: container-optimized.

rithms (as described in §4). The configurations for previous work are the default in their papers (e.g., number of queues) unless otherwise stated.

RIPQ+ is based on RIPQ [38], a container-based flash caching framework to approximate several caching algorithms that use queue structures. As a block in a container is accessed, its ID is copied into a virtual container in RAM, and when a container is selected for eviction, any blocks referenced by virtual containers are copied forward. We adopt a modified version of RIPQ that handles overwrite operations, referred to as RIPQ+ [23]. We use the segmented-LRU algorithm [17] and a container size of 2MB with 8 insertion points.

Pannier [23] is a container-based flash caching mechanism that identifies divergent (heterogeneous) containers where blocks held therein have highly varying access patterns. Pannier uses a priority-based survival queue to rank containers based on their survival time, and it selects a container for eviction that has either reached the end of its survival time or is the least-recently used in a segmented-LRU structure. During eviction, frequently accessed blocks are copied forward into new containers. Pannier also uses a multi-step credit-based throttling scheme to ensure flash lifespan.

6 Evaluation

In §6.1 we evaluate a number of caching algorithms with respect to RHR and EPBPD. We also evaluate the contribution of various techniques on the improvement in endurance. This is followed in §6.2 by a sensitivity analysis of some of the parameters, the use of the write buffer and overprovisioning. We then consider tradeoffs that improve endurance at a cost in RHR (§6.3).

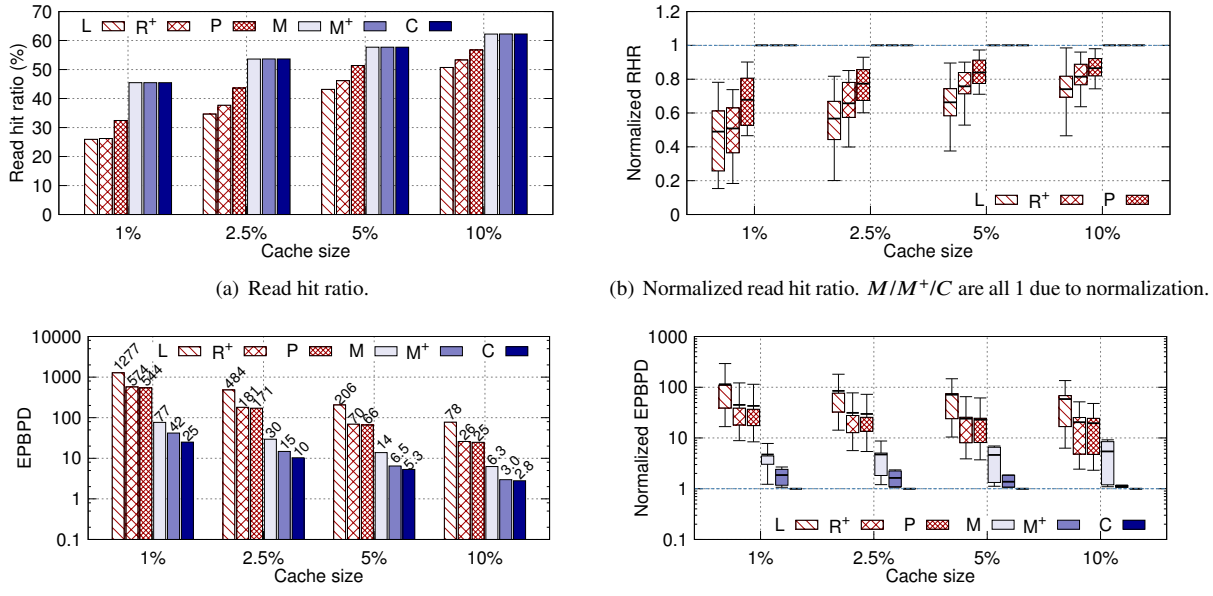
6.1 Comparing Caching Algorithms

We first compare the three online algorithms from Table 2 with the RHR-maximizing offline algorithms: MIN, M^+ and C. We focus initially on **O1**, minimizing erasures subject to a maximal read hit ratio. Figure 5 shows the RHR and EPBPD results across all 34 traces, while varying the cache size for each trace.

In Figure 5(a), we see that LRU (the left bar in each set) obtains the lowest hit rate because it has neither future knowledge nor a sophisticated cache replacement algorithm. RIPQ+ has about the same RHR as LRU; its benefits arise in reducing erasures rather than increasing hit rate. Pannier achieves up to 26% improvement in RHR over LRU and RIPQ+. By leveraging future knowledge, MIN, M^+ and C achieve the (identical) highest hit ratio, which improves upon Pannier by 9.7%–40%. The gap is widest for the smallest cache sizes. Figure 5(b) shows the range of the normalized RHR (normalized against the best-case C), with a similar trend as shown in Figure 5(a).

Figure 5(c) and 5(d) show that online algorithms incur the most erasures. Pannier performs slightly better than RIPQ+ due to the divergent container management and more efficient flash space utilization. Though it is not explicitly erasure-aware, MIN saves up to 86% of erasures compared to Pannier. This is because with perfect future knowledge, MIN can decide not to insert blocks that would never be referenced or whose next reference is a write. This implicit admission control mechanism results in significantly fewer flash erasures and higher RHR. M^+ further reduces erasures by 31% compared to MIN, because M^+ avoids inserting blocks that would be evicted before being accessed and uses TRIM to avoid copying blocks during GC if they will not be rereferenced. Variation does exist, as shown in Figure 5(d): the 10% and 25%-ile (the lower bound of box and whiskers) breakdown of M^+ are closer to 1 than those of MIN; the lower bounds never reach below 1 while the upper bounds (75% and 90%-ile) are far above, especially for small cache sizes. At small cache sizes, the container-optimized offline heuristic, C, further improves on M^+ by 40% by lowering GC costs.

One way to understand the differences among MIN, M^+, and C is to view the contributions of the individual improvements. The cumulative fraction of erasures saved by the techniques used by M^+ or C (summarized in Table 1) is depicted in Figure 6, normalized against the erasures used by MIN. Surprisingly, we find that simple techniques such as R_1 and TRIM have a greater impact on reducing erasures compared to more advanced techniques such as the container-optimized E.



(a) Read hit ratio.

(b) Normalized read hit ratio. $M/M^+/C$ are all 1 due to normalization.

(c) EPBPD.

(d) Normalized EPBPD.

Figure 5: RHR and EPBPD for various online/offline caching algorithms and sizes. EPBPD is shown on a log scale, with values above the bars to ease comparisons. The box-and-whisker plots in (b) and (d) show the {10, 25, average, 75, 90}%-ile breakdown of the normalized RHR and EPBPD, respectively; each is normalized to that of the best-case C.

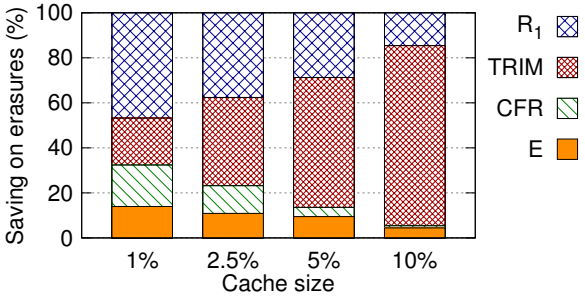


Figure 6: Fraction of erasures saved in different stages. R_1: never inserting blocks that will be evicted before being read. TRIM: removing blocks that would not be reread at FTL layer, CFR: avoiding wasted CFs, E: packing blocks in write buffer using eviction timestamp.

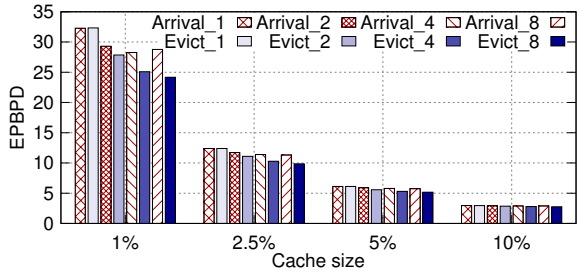
- The top component in each stacked bar (**R_1**) shows the relative improvement from preventing the insertion of blocks that will be evicted without being referenced; for the smallest cache, this accounts for about half the overall improvement from all techniques, but it is a relatively small improvement for the largest cache.
- For MIN, using **TRIM** to avoid the FTL copying of blocks that will not be reaccessed has an enormous contribution for the largest cache (˜80% of all erasures eliminated), but it is a much smaller component of the savings from the smallest caches. Note that we convert MIN to M^+ by avoiding (1) unread blocks due to eviction and (2) FTL GC for unneeded blocks.
- Adding a check for blocks that are copied forward (**CFR**) but then evicted without being rereferenced has a moderate (10%) impact on the smallest cache, but little impact on the largest. This is done only for C, as the copy-forwarding within the FTL for M^+ occurs in a separate component.
- Using 4 write buffers and grouping by eviction timestamp (**E**) has a similar effect to **CFR** on smaller caches and a nontrivial improvement for larger ones.

6.2 Sensitivity Analysis

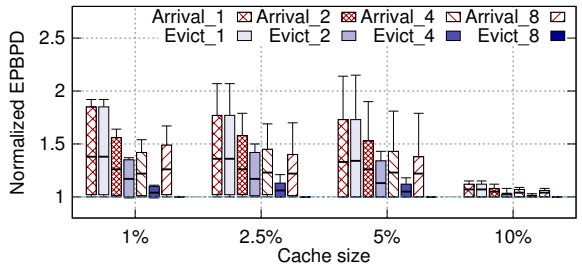
Thus far we have focused on default configurations. Here we compare the impact of different in-memory write buffer designs and sizes on erasures. Then we examine the impact of over-provisioned capacity.

Impact of Consolidating Blocks

We study the impact of consolidating blocks with similar eviction timestamps into the same containers and the impact of sizing the write buffer. Figure 7 plots the average EPBPD and variation across all traces, as a function of policy and write buffer size, grouped by different cache sizes. All experiments are performed using C and give the same optimal RHR. By default C uses the priority-based queue structure as the in-memory write buffer, which is filled using the eviction timestamp indicating when, using M^+, the block would be evicted from the cache. The write buffer, once full, is dispersed into containers that are written into flash. For comparison purposes we implemented a FIFO-queue-based write buffer, where the blocks are simply sorted based on their

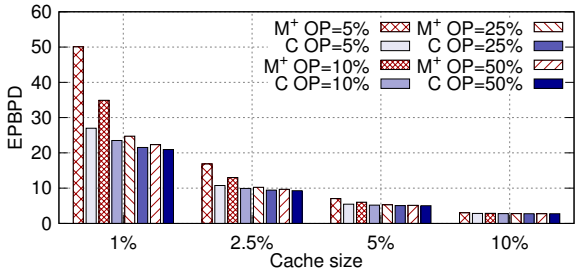


(a) EPBPD.

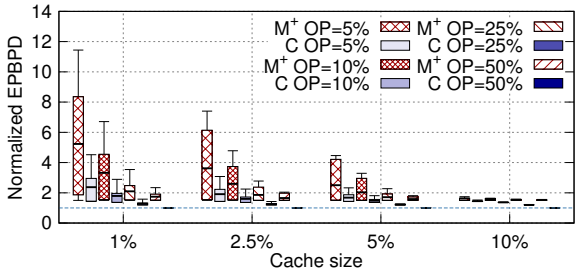


(b) Normalized EPBPD.

Figure 7: Impact of consolidating blocks based on eviction timestamp, and trade-offs in varying the size of the in-memory write buffer (a multiple of containers); `Arrival_2` means packing the blocks into the write buffer (2-container worth of capacity) based on their arrival time, `Evict_4` means packing the blocks into the 4-container-sized write buffer based on the eviction timestamp. The box-and-whisker plot in (b) shows the {10, 25, average, 75, 90}%-ile breakdown of the EPBPD, normalized to that of `Evict_8`.



(a) EPBPD.



(b) Normalized EPBPD.

Figure 8: Trade-offs in over-provisioned (OP) space. The box-and-whisker plot shows normalized EPBPD against `C OP=50%`.

arrival timestamp.[4] There is no difference in EPBPD with a queue size of 1 container, because blocks cannot be sorted by eviction timestamp with a single open container in DRAM. Increasing the write buffer size, we observe a reduction in erasures with `Evict`. This effect is more pronounced with a bigger write buffer queue. For example, for the 2.5% cache size, `Evict_2` reduces the EPBPD by 5% compared to `Arrival_2`; but this EPBPD differential increases to 13% for an 8-container write buffer. This is because a bigger `Evict` write buffer results in less fragmentation due to evict-pending blocks in containers stored in flash. The trend shown in average consistently matches that of the individual variation in Figure 7(b). Interestingly, `Arrival_8` with a 1% cache size yields a slightly higher EPBPD than that of `Arrival_4`. This is because the fraction of data copied forward internally (due to GC) is higher when using a relatively small cache and a large write buffer. We observed that while the 4 least populated containers generally were sufficiently "dead" to benefit from GC, the next 4 (5^{th}–8^{th} least populated) containers hold significantly more live data than the first four when collected in a batch; this increases CF significantly.

[4]Packing blocks based on LBA or whether clean (newly inserted) or dirty (invalidated due to overwrite), yields no impact on erasures or WFUE scores for the offline heuristics.

Impact of Over-provisioned Capacity

Next, we analyze the impact of over-provisioned (OP) capacity on erasures. Figure 8 shows the average EPBPD when varying the over-provisioned space between 5% and 50% for different cache sizes. As in the previous experiment, we omit results of RHR, which are unaffected by overprovisioning. We classify the results into block-level (M^+) and container-optimized (C) approaches, which are interspersed and grouped by the amount of over-provisioned capacity. (Thus, for a given capacity, it is easy to see the impact of the container-optimized approach.)

For both groups, a larger OP space results in lower EPBPD, because the need for GC to reclaim new space becomes less urgent than a flash equipped with relatively smaller OP space. This effect is more significant for M^+, as M^+ manages data placement in a container-oblivious manner; this results in more GCs, which in turn cause larger write amplification (more internal writes at the FTL level). We observe that for a 1% cache size, C with 10% OP incurs fewer erasures than M^+ with 25% OP. This is because C consolidates blocks that would be removed at roughly the same time, resulting in significantly fewer internal (CF) flash writes. C also avoids CF of most blocks that get evicted before reaccess. With the largest cache, however, the relative benefit from additional overprovisioning is nominal. Again, Figure 8(b) demonstrates that the variation across traces exists; and

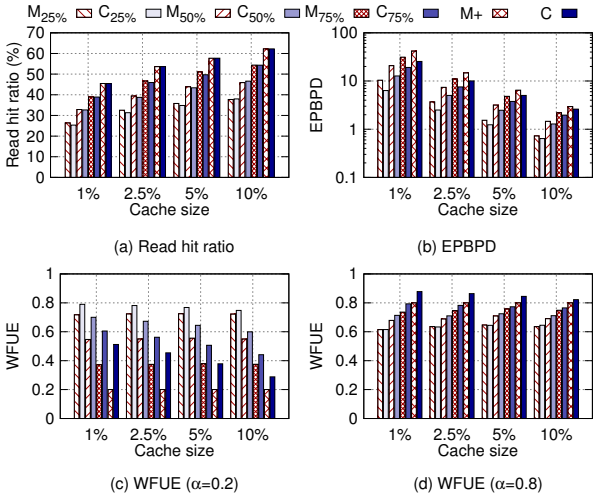


Figure 9: Trade-offs when limiting erasures: WFUE as a function of algorithm, EPBPD quota, cache size, and α weights.

it is just a matter of how many more erasures each trace incurs, compared to the best case.

6.3 Throttling Flash Writes

Thus far the evaluation has focused on **O1**, optimizing first for RHR and second for EPBPD. If erasures are more important, we can limit flash writes to reduce EPBPD at some cost to hit rate. Recall that **O2** tries to maximize RHR subject to a specific limit, whereas **O3** tries to optimize WFUE given a particular weight of the importance of RHR relative to EPBPD.

Figure 9 demonstrates the effect of C_T, which uses the admission control logic described in §3.5 to meet a specific EPBPD limit. It removes insertions with the least impact (i.e., blocks with least number of read hits) on RHR to meet the endurance goal. Figure 9 shows the results averaged across all traces when varying the EPBPD quota for a trace from 25%–75% of the EPBPD necessary for M or C respectively; this represents a reasonable range of user requirements on flash lifespan in real-world cases. For each cache size there are eight bars, with pairs of M_T and C_T algorithms as the threshold varies from 25% to 100% of total erasures. (The limits for M_T and C_T are set differently, since the maximum values vary.)

We observe in Figure 9(a) that for big cache sizes (5% and 10%) the RHR loss is about 39% for the 25% EPBPD quota. The gap reduces to 13% for the 75% EPBPD quota. As expected and shown in Figure 9(b), overall EPBPD decreases as the threshold is lowered, while C_T moderately improves upon M_T.

For WFUE, one question is what an appropriate weight α would be. Figures 9(c) and (d) plot the same algorithms but report WFUE using $\alpha = 0.2$ and $\alpha = 0.8$ (this prioritizes erasures and hit rates respectively). With $\alpha = 0.2$, the erasure savings dominate. Hence, $M_{25\%}$ and $C_{25\%}$ achieve the highest WFUE scores while M^+ and C see lower ones. Prioritizing RHR gives M^+ and C the highest WFUE across all variants. C_T consistently outperforms the corresponding M_T because it can avoid most wasted CFs and because it groups by eviction timestamp (see §6.1).

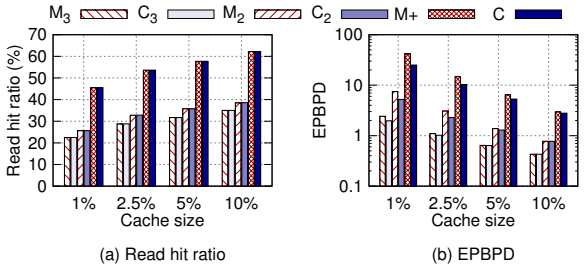


Figure 10: Trade-offs in read hit based insertion removals.

Figure 10 shows the effect of limiting flash insertions to blocks that are reread $\geq N$ times. It plots M^+ (which is equivalent to M_1), M_2, and M_3, as well as the corresponding container-optimized C algorithms. Results for RHR and EPBPD are averaged across all 34 traces. For small cache sizes (1% and 2.5%), C_2 loses an average of 41% of the RHR (Figure 10(a)), but it gets about a 79% savings in EPBPD (Figure 10(b)). Prioritizing erasures and RHR shows similar trends as the WFUE results in read hit based insertion removal tests (Figures 9(c) and (d)), hence are omitted due to space constraints.

7 Conclusion and Future Work

While it is challenging to optimize for both RHR and endurance (represented by EPBPD) simultaneously, we have presented a set of techniques to improve endurance while keeping the best possible RHR, or conversely, trade off RHR to limit the impact on endurance. In particular, our container-optimized heuristic can maintain the maximal RHR while reducing flash writes caused by garbage collection; we see improvements of 55%–67% over MIN and 6%–40% over the improved M^+, which avoids many wasted writes and uses TRIM to reduce GC overheads. Another important finding in our study indicates that simple techniques such as R_1 and TRIM provide most of the benefit in minimizing erasures. Alternatively, the flash writes can be limited to those that are rereferenced a minimum number of times. We define a new metric, Weighted Flash Usage Effectiveness, which uses the offline best case as a baseline to evaluate tradeoffs between RHR and EPBPD quantitatively.

In the future, we would like to investigate the complexity of the various algorithms (we believe them to be NP-hard). Exploring approaches to improving online flash caching algorithms is also part of our future work. We are particularly interested in heuristics to trade off one cache hit against another to further reduce cache writes without impacting RHR.

Acknowledgments

We are grateful to our shepherd, Dan Tsafrir, as well as the anonymous reviewers, for their valuable comments and suggestions that helped improve the paper. We would also like to thank Dan Arnon, Cheng Li, Stephen Manley, Darren Sawyer, and Kevin Xu for their general feedback, and Prof. Sanjeev Arora from Princeton University, and his students, Holden Lee, Fermi Ma, Karen Singh, and Cyril Zhang, for discussions on algorithm complexities. This work was supported in part by NSF award CNS-1149232.

References

[1] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. Page replacement for general caching problems. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA'99)*, 1999.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966.

[3] Avraham Ben-Aroya and Sivan Toledo. Competitive analysis of flash memory algorithms. *ACM Trans. Algorithms*, 7(2):1–37, March 2011.

[4] M. Brehob, S. Wagner, E. Torng, and R. Enbody. Optimal replacement is np-hard for nonstandard caches. *Computers, IEEE Transactions on*, 53(1):73–76, Jan 2004.

[5] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'05)*, 2005.

[6] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. Caching is hard—even in the fault model. *Algorithmica*, 63(4):781–794, 2012.

[7] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, September 2010.

[8] Facebook Flashcache. https://github.com/facebook/flashcache.

[9] Ziqi Fan, David HC Du, and Doug Voigt. H-arc: A non-volatile memory based cache policy for solid state drives. In *Proceedings of the 30th Mass Storage Systems and Technologies Symposium (MSST'14)*. IEEE, 2014.

[10] Binny S. Gill. On multi-level exclusive caching: Offline optimality and why promotions are better than demotions. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008.

[11] Chris Gniady, Ali R. Butt, and Y. Charlie Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th USENIX Conference on Symposium on Operating Systems Design & Implementation (OSDI'04)*, 2004.

[12] GNU Parallel. http://www.gnu.org/software/parallel/.

[13] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. Flash caching on the storage client. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, 2013.

[14] L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd. Index register allocation. *J. ACM*, 13(1):43–61, January 1966.

[15] Sai Huang, Qingsong Wei, Jianxi Chen, Cheng Chen, and Dan Feng. Improving flash-based disk cache with lazy adaptive replacement. In *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST'13)*, May 2013.

[16] Jffs2: The journalling flash file system, version 2. https://sourceware.org/jffs2/.

[17] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[18] S. Kavalanekar, B. Worthington, Qi Zhang, and V. Sharda. Characterization of storage workload traces from production windows servers. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*, Sept 2008.

[19] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. Write policies for host-side flash caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, February 2013.

[20] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, February 2015.

[21] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-managed flash. In *Proceedings of the 14th USENIX Confer-*

ence on File and Storage Technologies (FAST'16), February 2016.

[22] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, June 2014.

[23] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th International Middleware Conference (Middleware'15)*, December 2015.

[24] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. Nvmkv: A scalable and lightweight flash aware key-value store. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*, June 2014.

[25] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[26] Lyle A. McGeoch and Daniel D. Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, 1991.

[27] Micron MLC SSD Specification. http://www.micron.com/products/nand-flash, 2013.

[28] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST'08)*, February 2008.

[29] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, February 2012.

[30] Yongseok Oh, Eunjae Lee, Choulseung Hyun, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Enabling cost-effective flash based caching with an array of commodity ssds. In *Proceedings of the 16th Annual Middleware Conference (Middleware'15)*, December 2015.

[31] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*. ACM, 2014.

[32] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*, June 2014.

[33] Samsung Server SSD Specification. http://www.samsung.com/serverssd/, 2015.

[34] SanDisk SATA Solid State Drives. http://www.sandisk.com/enterprise/sata-ssd/, 2015.

[35] Ricardo Santana, Steven Lyons, Ricardo Koller, Raju Rangaswami, and Jason Liu. To ARC or not to ARC. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, July 2015.

[36] Mohit Saxena, Michael M. Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, 2012.

[37] Hyong Shim, Philip Shilane, and Windsor Hsu. Characterization of incremental data changes for efficient data protection. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*, 2013.

[38] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq: Advanced photo caching on flash for facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, February 2015.

[39] Olivier Temam. Investigating optimal local memory performance. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*. ACM, 1998.

[40] TRIM Specification. ATA/ATAPI Command Set- 2 (ACS-2). http://www.t13.org/.

[41] Yaffs (Yet Another Flash File System). http://www.yaffs.net/.

[42] Jingpei Yang, Ned Plasson, Greg Gillis, and Nisha Talagala. Hec: Improving endurance of high performance flash-based cache devices. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*. ACM, 2013.

[43] Yifeng Zhu and Hong Jiang. Race: A robust adaptive caching strategy for buffer cache. *IEEE Trans. Comput.*, 57(1):25–40, January 2008.

Unlocking Energy

Babak Falsafi *Rachid Guerraoui* *Javier Picorel* *Vasileios Trigonakis* *
{first.last}@epfl.ch
EcoCloud, EPFL

Abstract

Locks are a natural place for improving the energy efficiency of software systems. First, concurrent systems are mainstream and when their threads synchronize, they typically do it with locks. Second, locks are well-defined abstractions, hence changing the algorithm implementing them can be achieved without modifying the system. Third, some locking strategies consume more power than others, thus the strategy choice can have a real effect. Last but not least, as we show in this paper, improving the energy efficiency of locks goes hand in hand with improving their throughput. It is a win-win situation.

We make our case for this throughput/energy-efficiency correlation through a series of observations obtained from an exhaustive analysis of the energy efficiency of locks on two modern processors and six software systems: Memcached, MySQL, SQLite, RocksDB, HamsterDB, and Kyoto Kabinet. We propose simple lock-based techniques for improving the energy efficiency of these systems by 33% on average, driven by higher throughput, and without modifying the systems.

1 Introduction

For several decades, the main metric to measure the efficiency of computing systems has been *throughput*. This state of affairs started changing in the past few years as *energy* has become a very important factor [17]. Reducing the power consumption of systems is considered crucial today [26, 30]. Various studies estimate that datacenters have contributed over 2% of the total US electricity usage in 2010 [36], and project that the energy footprint of datacenters will double by 2020 [1].

Hardware techniques for reducing energy consumption include clock gating [38], power gating [52], as well as voltage and frequency scaling [28, 56]. Software techniques include approximation [16, 27, 57], consolidation [18, 21], energy-efficient data structures [23, 32], fast active-to-idle switching [44, 45], power-aware schedulers [48, 55, 59], and energy-oriented compilers [63, 64]. Basically, those techniques require changes in hardware, installing new schedulers or runtime systems, or even rebuilding the entire system.

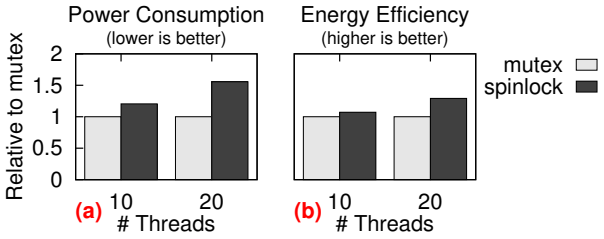


Figure 1: Power consumption and energy efficiency of `CopyOnWriteArrayList` with mutex and spinlock.

We argue that there is an effective, complementary approach to saving energy: Optimizing *synchronization*, specifically its most popular form, namely *locking*. The rationale is the following. First, concurrent systems are now mainstream and need to synchronize their activities. In most cases, synchronization is achieved through locking. Hence, designing locking schemes that reduce energy consumption can affect many software systems. Second, the lock abstraction is well defined and one can usually replace the algorithm implementing it without any modification to the rest of the system. Third, the choice of the locking scheme can have a significant effect on energy consumption. Indeed, the main consequence of synchronization is having some threads *wait* for one another–an opportunity for saving energy.

To illustrate this opportunity, consider the average power consumption of two versions of a `java.util.concurrent.CopyOnWriteArrayList` [6] stress test over a long-running execution–Figure 1(a). The two versions differ in how the lock handles contention: Mutexes use *sleeping*, while spinlocks employ *busy waiting*. With sleeping, the waiting thread is put to sleep by the OS until the lock is released. With busy waiting, the thread remains active, polling the lock until the lock is finally released. Choosing sleeping as the

*Author names appear in alphabetical order.

waiting strategy brings up to 33% benefits on power. Hence, as we pointed out, the choice of locking strategy can have a significant effect on power consumption.

Accordingly, privileging sleeping with mutex locks seems like the systematic way to go. This choice, however, is not as simple as it looks. What really matters is not only the power consumption, but the amount of energy consumed for performing some work, namely *energy efficiency*. In the Figure 1 example, although the spinlock version consumes 50% more power than mutex, it delivers 25% higher energy efficiency (Figure 1(b)) for it achieves twice the throughput. Hence, indeed, locking is a natural place to look for saving energy. Yet, choosing the best lock algorithm is not straightforward.

To finalize the argument that optimizing locks is a good approach to improve the energy efficiency of systems, we need locks that not only reduce power, but also do not hurt throughput. Is that even possible?

We show that the answer to this question is positive. We argue for the POLY[1] conjecture: *Energy efficiency and throughput go hand in hand in the context of lock algorithms*. POLY suggests that we can optimize locks to improve energy efficiency without degrading throughput; the two go hand in hand. Consequently, we can apply prior throughput-oriented research on lock algorithms almost as is in the design of energy-efficient locks as well.

We argue for our POLY conjecture through a thorough analysis of the energy efficiency of locks on two modern Intel processors and six software systems (i.e., Memcached, MySQL, SQLite, RocksDB, HamsterDB, and Kyoto Kabinet). We conduct our analysis in three layers. We start by analyzing the hardware and software artifacts available for synchronization (e.g., pausing instructions, the Linux `futex` system calls). Then, we evaluate optimized variants of lock algorithms in terms of throughput and energy efficiency. Finally, we apply our results to the six software systems. We derive from our analysis the following observations that underlie POLY:

Busy waiting inherently hurts power consumption. With busy waiting, the underlying hardware context remains active. On Intel machines, for example, it is not practically feasible to reduce the power consumption of busy waiting. First, there is no power-friendly pause instruction to be used in busy-wait loops. The conventional way of reducing the cost of these loops, namely the x86 `pause` instruction, actually increases power consumption. Second, the `monitor/mwait` instructions require kernel-level privileges, thus using them in user space incurs high overheads. Third, traditional DVFS techniques for decreasing the voltage and frequency of the cores (hence lowering their power consumption) are too coarse-grained and too slow to use. Consequently, the power consumption of busy waiting can simply not be reduced. The only way is to look into sleeping.

Sleeping can indeed save power. Our Xeon server has approximately 55 Watts idle power and a max total power consumption of 206 Watts. Once a hardware context is active, it draws power, regardless of the type of work it executes. We can save this power if threads are put to sleep while waiting behind a busy lock. The OS can then put the core(s) in one of the low-power idle states [5]. Furthermore, when there are more software threads than hardware contexts in a system, sleeping is the only way to go in locks, because busy waiting kills throughput.

However, going to sleep hurts energy efficiency. The `futex` system call implements sleeping in Linux and is used by pthread mutex locks. In most realistic scenarios, the `futex`-call overheads offset the energy benefits of sleeping over busy waiting, if any, resulting in worse energy efficiency. Additionally, the spin-then-sleep policy of mutex is not tuned to account for these overheads. The mutex spins for up to a few hundred cycles before employing `futex`, while waking up a sleeping thread takes at least 7000 cycles. As a result, it is common that a thread makes the costly `futex` call to sleep, only to be immediately woken up, thus wasting both time and energy. We design MUTEXEE, an optimized version of mutex that takes the `futex` overheads into account.

Thus, some threads have to go to sleep for long. An unfair lock can put threads to sleep for long periods of time in the presence of high contention. Doing so results in lower power consumption, as fewer threads (hardware contexts) are active during the execution. In addition, lower fairness brings (i) better throughput, as the contention on the lock is decreased, and (ii) higher tail latencies, as the latency distribution of acquiring the lock might include some large values.

Overall, on current hardware, every power trade-off is also a throughput and a latency trade-off (motivating the name POLY[1]): (i) sleeping vs. busy waiting, (ii) busy waiting with vs. without DVFS or `monitor/mwait`, and (iii) low vs. high fairness.

Interestingly, in our quest to substantiate POLY, we optimize state-of-the-art locking techniques to increase the energy efficiency of our considered systems. We improve the systems by 33% on average, driven by a 31% increase in throughput. These improvements are either due to completely avoiding sleeping using spinlocks, or due to reducing the frequency of sleep/wake-up invocations using our new MUTEXEE scheme.

We conduct our analysis on two modern Intel platforms as they provide tools (i.e., RAPL interface [4]) for accurately measuring the energy consumption of the processor. Still, we believe that POLY holds on most modern

[1] POLY stands for "Pareto optimality in locks for energy efficiency."

multi-cores. On the one hand, without explicit hardware support, busy waiting on any multi-core exhibits similar behavior. On the other hand, `futex` implementations are alike regardless of the underlying platform, thus the overheads of sleeping will always be significant. However, should the hardware provide adequate tools for fine-grained energy optimizations in software, POLY might need to be revised. We discuss the topic further in §8.

In summary, the main contributions of this paper are:

- The POLY conjecture, stating that we can simply, yet effectively optimize lock-based synchronization to improve the energy efficiency of software systems.
- An extensive analysis of the energy efficiency of locks. The results of this analysis can be used to optimize lock algorithms for energy efficiency.
- Our lock libraries and benchmarks, available at: `https://lpd.epfl.ch/site/lockin`.
- MUTEXEE, an improved variant of pthread mutex lock. MUTEXEE delivers on average 28% higher energy efficiency than mutex on six modern systems.

It is worth noting that POLY might not seem surprising to a portion of the multi-core community. Yet, we believe it is important to clearly state POLY and quantify through a thorough analysis the reasons why it is valid on current hardware. As we discuss in §8, our results have important software and hardware ramifications.

The rest of the paper is organized as follows. In §2, we recall background notions regarding synchronization and energy efficiency. We describe our target platforms in §3 and explore techniques for reducing the power of synchronization in §4. We analyze in §5 the energy efficiency of locks and we use our results to improve various software systems in §6. We discuss related work in §7, and we conclude the paper in §8.

2 Background and Methodology

Lock-based Synchronization. Locks ensure mutual exclusion; only the holder of the lock can proceed with its execution. The remaining threads wait until the holder releases the lock. This waiting is implemented with either *sleeping* (*blocking*), or *busy waiting* (*spinning*) [49].

With sleeping, the thread is put in a per-lock wait queue and the hardware context is released to the OS. When the lock is released, the OS might wake up the thread. With busy waiting, threads remain active, polling the lock in a spin-wait loop.

Sleeping is employed by the pthread mutex lock (MUTEX). MUTEX builds on top of `futex` system calls, which allow a thread to wait for a value change on an address. MUTEX might first perform busy waiting for a limited amount of time and if the lock cannot be acquired, the thread makes the `futex` call.

The locks which use busy waiting are called *spinlocks*. There are several spinlock algorithms, such as test-and-set (TAS), test-and-test-and-set (TTAS), ticket-lock (TICKET) [46], MCS (MCS) [46], and CLH (CLH) [22]. Spinlocks mostly differ in their busy-waiting implementation. For example, TAS spins with an atomic operation, continuously trying to acquire the lock (*global spinning*). In contrast, all other spinlocks spin with a load until the lock becomes free and only then try to acquire the lock with an atomic operation (*local spinning*).

Energy Efficiency of Software. *Energy efficiency* represents the amount of work produced for a fixed amount of energy and can be defined as *throughput per power* (TPP, $\#operation/Joule$). Higher TPP represents a more energy-efficient execution. We use the terms energy efficiency and TPP interchangeably. Alternatively, energy efficiency can be defined as the energy spent on a single operation, namely *energy per operation* (EPO, $Joule/operation$). Note that TPP = 1/EPO.

Experimental Methodology. We prefer TPP over EPO because both throughput and TPP are "higher-is-better" metrics. Recent Intel processors include the RAPL [4] interface for accurately measuring energy consumption. RAPL provides counters for measuring the cores', package, and DRAM energy. We use these energy measurements to calculate average power. Our microbenchmark results are the median of 11 repetitions of 10 seconds. When we vary the number of threads, we first use the cores within a socket, then the cores of the second socket, and finally, the hyper-threads.

3 Target Platforms

We describe our two target platforms and then estimate their maximum power consumption.

Platforms. We use two modern Intel processors:

Name	Type	#Cores	L1	L2	LLC	Mem	TDP
Xeon	server	10	32KB	256KB	25MB	128GB	115W
Core-i7	desktop	4	32KB	256KB	8MB	16GB	77W

In the interest of space and clarity of explanation, we focus in the paper on the results of our server. Note that the results on Core-i7 are in accordance with the ones on Xeon. Our server is a two-socket Intel Ivy Bridge (E5-2680 v2), henceforth called *Xeon*. Xeon runs on frequencies scaling from 1.2 to 2.8 GHz due to DVFS and uses the Linux kernel 3.13 and glibc 2.13. Our desktop is an Intel Core i7 (Ivy Bridge–3770K) processor, henceforth called *Core-i7*. Core-i7 runs on frequencies scaling from 1.6 to 3.5 GHz due to DVFS and runs the Linux kernel 3.2 and glibc 2.15. Both platforms have two hardware threads per-core (hyper-threads in Intel's terminology). Intel turbo boost is disabled for all the experiments.

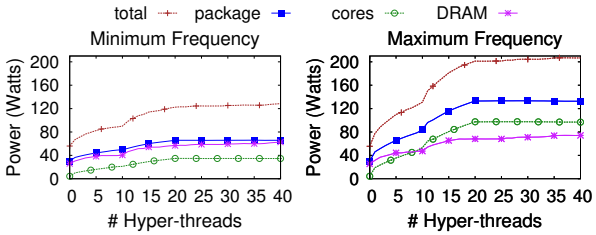


Figure 2: Power-consumption breakdown on Xeon.

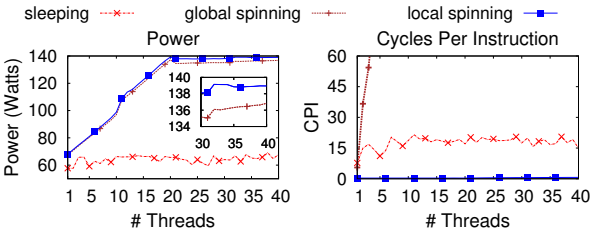


Figure 3: Power consumption and CPI while waiting.

3.1 Estimating Max Power Consumption

We estimate the maximum power that Xeon can consume, using a memory-intensive benchmark that consists of threads sequentially accessing large chunks of memory from their local node. Figure 2 depicts the total power and the power of different components on Xeon, depending on the number of active hyper-threads and the voltage-frequency (VF) setting.

Idle Power Consumption. The 0-thread points represent the idle power consumption, which accounts for the static power in cores and caches, and DRAM background power, and is the power that is consumed when all cores are inactive.[2] In both min and max frequency settings the total idle power is 55.5 Watts as the VF setting only affects the active power.

Power of Active Cores. Activating the first core of a socket is more expensive than activating any subsequent due to the activation of the uncore package components. In particular, it costs 6.4 and 13.6 Watts in package power on the min and max VF settings, respectively. The second core costs 2.3 and 5.6 Watts. We perform more experiments (not shown in the graphs) with data sets that fit in L1, L2, and LLC. The results show that the package power is not vastly reduced on any of these workloads, indicating that once a core is active, the core consumes a certain amount of power that cannot be avoided.

Attribution of Power to Cores, Package, and Memory. Notice the breakdown of total power to package/core[3] and DRAM power. DRAM power has a smaller dynamic range than package and core power. On the max VF setting, DRAM power ranges from 25 to 74 Watts, while the range of package power is from 30 to 132 Watts, and core power from 4 up to 96 Watts.

Implications. The power consumption of Xeon ranges from 55 up to 206 Watts. Out of the 206 Watts, 74 Watts are spent on the DRAM memory. Locks are typically transferred within the processor by the cache-coherence protocol, thus limiting the opportunities for reducing power to package power (30-132 Watts). Additionally, once a core is active, the core draws power, regardless of the type of work performed. Consequently, the opportunity for reducing power consumption in software is relatively low and mostly has to do with (i) using fewer cores, by, for example, putting threads to sleep, or (ii) reducing the frequency of a core.

[2]Still, the OS briefly enables a few cores during the measurements.
[3]The package power includes the core power.

4 Reducing Power in Synchronization

In this section, we evaluate the costs of busy waiting and sleeping, and examine different ways of reducing them.

4.1 Power: The Price of Busy Waiting

We measure the total power consumption of the three main waiting techniques (i.e., sleeping, global spinning, and local spinning–see §2) when all threads are waiting for a lock that is never released. Figure 3 shows the power consumption and the cycles per instruction (CPI). CPI represents the average number of CPU cycles that an instruction takes to execute.

Two main points stand out. First, in this extreme scenario, sleeping is very efficient because the waiting threads do not consume any CPU resources. Second, local spinning consumes up to 3% more power than global spinning. This behavior is explained by the CPI graph: Global spinning performs atomic operations on the shared memory address of the lock, resulting in a very high CPI (up to 530 cycles). In local spinning, every thread executes an L1 load each cycle, whereas, in global spinning, storing over coherence occurs once the atomic operation is performed, each 530 cycles on average.

4.2 Reducing the Price of Busy Waiting

We reduce the power consumption of busy waiting in different ways: (i) we examine various ways of pausing in spin-wait loops, (ii) we employ DVFS, and (iii) we use `monitor/mwait` to "block" the waiting threads.

Pausing Techniques. Busy waiting with local spinning is power hungry, because threads execute with low CPI. Hence, to reduce power, we must increase the loop's CPI. We take several approaches to this end (Figure 4).

Any instruction, such as a `nop`, that the out-of-order core can hide, cannot reduce the power of the spin loop. According to Intel's Software Developer's Manual [4], "*Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.*" A `pause` (*local-pause*) increases CPI to 4.6. However, not only

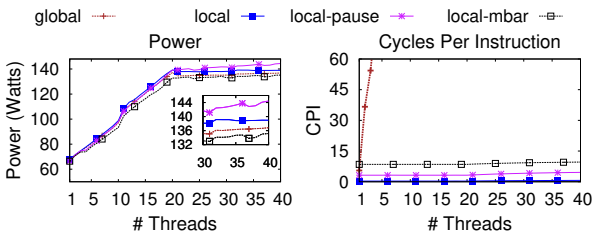


Figure 4: Power consumption and CPI while spinning.

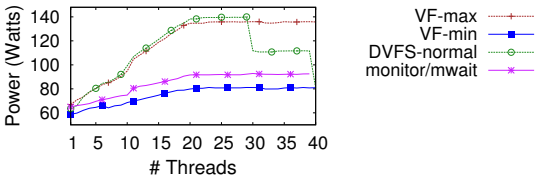


Figure 5: Power consumption of busy waiting using DVFS and `monitor/mwait`.

does it not "greatly reduce" power, but it even increases the power consumption by up to 4%.[4]

In general, the reason behind the very low CPI of local spinning is the aggressive execution mechanisms of modern processors that allow instructions to execute both speculatively and out of the program order. This results in one out of three of the retired operations being a memory load (the other two are a test and a conditional jump). Without appropriate pausing, the spin loop retires one memory load per cycle.

A way to avoid the speculative execution of the load is to insert a full, or a load, memory barrier. That way, the loads only execute once the previous load retires and the instructions that depend on it, test and jump, are stalled as well. The results (*local-mbar*) show that the barrier reduces the power consumption of local spinning to the point that becomes less expensive than global spinning (*global*). Additionally, *local-mbar* consumes up to 7% less power than *local-pause*. It is worth noting that *local-mbar* consumes less power than *local-pause* even for low thread counts (e.g., 5% on 10 threads). In the rest of the paper, we use a memory barrier for pausing in spin loops.

Dynamic Voltage and Frequency Scaling (DVFS). An intuitive way of lowering the power consumption of an active core is to reduce the voltage-frequency (VF) point via DVFS (see §3). Figure 5 shows that spinning on *VF-min* consumes up to 1.7x less power than on the *VF-max* setting. Still, DVFS is currently impractical for dynamically reducing power in busy waiting.

First, to trigger the VF change with DVFS, we need to write on a certain per-hardware context file of the `/sys/devices` directory (more details about DVFS can be found in [62]). Hence, the VF-switch operation is slow: We measure that it takes 5300 cycles on Xeon. If DVFS is used while busy waiting, this overhead will be on the critical path when the lock is acquired and the thread must switch back to the maximum VF point.

Second, both hyper-threads of a physical core share the same VF setting–the higher of the two. If a hyper-thread lowers its VF setting, the action will have no effect unless the second hyper-thread has the same or lower VF setting. Consequently, using DVFS with hyper-threading is tricky, and as the *DVFS-normal* line shows, the power consumption drops only when both hyper-threads lower their VF points.

Monitor/mwait. The `monitor/mwait` [4] instructions allow a hardware context to block and enter an implementation-dependent optimized state while waiting for a store event. In detail, `monitor` allows a thread to declare a memory range to monitor. The hardware thread then uses `mwait` to enter an optimized state until a store is performed within the address range. Essentially, `mwait` implements sleeping in hardware and can be used in spin-wait loops: The hardware sleeps, yet the thread does not release its context.

These instructions require kernel privileges. We develop a virtual device and overload its *file_operations* functions to allow a user program to declare and wait on an address, similar to [14]. A thread can wake up others with a user-level store. However, threads pay the user-to-kernel switch and system-call overheads for waiting.

Figure 5 includes the power of busy waiting with `monitor/mwait`. These instructions can reduce power consumption over conventional spinning up to 1.5x. However, similarly to DVFS, using `monitor/mwait` has two shortcomings. First, `monitor/mwait` can be only used in kernel space. The overloaded file operation takes roughly 700 cycles. The best case wake-up latency from `mwait`, with just one core "sleeping," is 1600 cycles. In comparison, "waking up" a locally-spinning thread takes two cache-line transfers (i.e., 280 cycles). Second, programming with `monitor/mwait` on Intel processors can be elaborate and limiting. The `mwait` instruction blocks the hardware context until the thread is awaken. In oversubscribed environments (i.e., more threads than hardware contexts), `monitor/mwait` will likely exacerbate the "livelock" issues of spinlocks (see §6). Blocked threads might occupy most hardware contexts, thus preventing other threads from performing useful work.

Implications. Busy waiting drains a lot of power because cores execute at full speed. Neither of the two platforms provides sufficient tools for reducing power consumption in a systematic way. Pausing techniques, such as `pause`, can even increase the power of busy waiting.

[4] We speculate that one of the reasons for this increase in power is that `pause` gives a hint to the core to prioritize the other hyper-thread.

Techniques that can significantly reduce power, such as DVFS and `monitor/mwait`, are not designed for user-space usage as they require expensive kernel operations. Hence, sleeping is currently the only practical way of reducing the power consumption in locks.

4.3 Latency: The Price of Sleeping

In Linux, sleeping is implemented with `futex` system calls. A `futex`-sleep call puts the thread to sleep on a given memory address. A `futex` wake-up call awakes the first N threads sleeping on an address ($N = 1$ in locks). The `futex` calls are protected by kernel locks. In particular, the kernel holds a hash table (array) of locks and `futex` operations calculate the particular lock to use by hashing the address. Given that the array is large (approximately $256 * \#cores$ locks), the probability of false contention is low. However, operations on the same address (same MUTEX) do contend on kernel level.

We use a microbenchmark where two threads run in lock-step execution (synchronized at each round with barriers). One makes `futex`-sleep calls and the second makes wake-up calls on the same `futex`, after waiting for some time. A `futex`-sleep call (i.e., enqueuing behind the lock and descheduling the thread) takes around 2100 cycles.[5] This sleep latency is not necessarily on the critical path: The thread sleeps because the lock is occupied. However, the latency to wake up a thread and the one for the woken-up thread to be ready to execute are on the critical path. Figure 6 contains the wake-up call and the turnaround latencies, depending on the delay between the invocation of the sleep call and the wake-up call. The turnaround latency is the time from the wake-up invocation until the woken-up thread is running.[6]

The turnaround time is at least 7000 cycles and is higher than the wake-up call latency. Apart from the approximately 2700 cycles of the wake-up call, the woken-up thread requires at least 4000 more cycles before executing. Concretely, once the wake-up call finishes, the woken-up thread pays the cost of idle-to-active switching and the cost of scheduling.[7]

Figure 6 further includes two interesting points. First, for low delays between the two calls, the wake-up call is more expensive as it waits behind a kernel lock for the completion of the sleep call. Second, when the delay between the calls is very large (>600K cycles), the turnaround latency explodes, because the hardware context sleeps in a deeper idle state [41].

Finally, the results in Figure 6 use just two threads and thus represent the best-case latencies, with minimal or no contention at the kernel level. With more threads, a wake-up invocation is likely to contend with `futex` sleep calls, all serialized using a single kernel lock.

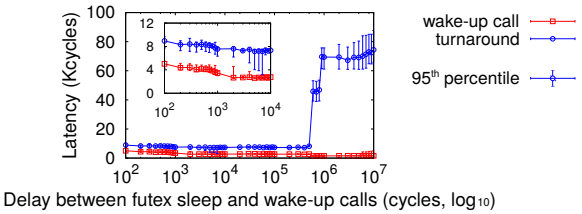


Figure 6: Latency of different `futex` operations.

Implications. `futex` operations have high latencies and consume energy, as a non-negligible number of instructions are executed. Handing over a lock with a `futex` wake-up call requires at least 7000 cycles. Even on rather lengthy critical sections (e.g., 10000 cycles), this latency is prohibitive; it almost doubles the execution time of the critical section. In this case, the energy benefits of sleeping will not easily compensate the performance losses. In short critical sections, invoking `futex` calls will have detrimental effects on performance.

4.4 Reducing the Price of Sleeping

Sleeping can save energy on long waiting durations. We estimate when sleeping reduces power consumption with two threads:

Period between wake-up calls (cycles)	1024	2048	4096	8192
Power (Watts)	72.03	69.18	68.75	68.02

The first thread sleeps on a location, while the second periodically wakes up the first thread. We vary the period between the wake-up invocations, which essentially represents the critical-section duration in locks. The results confirm that if a thread is woken up more frequently than the `futex`-sleep latency, power consumption is not reduced. The thread goes to sleep only to be immediately woken up by a concurrent wake-up call. When these "sleep misses" happen, we lose performance without any power reduction. Once the delay becomes larger than the sleep latency (i.e., approximately 2100 cycles on Xeon), we start observing power reductions.

Reducing Fairness. We show two problems with `futex`-based sleeping: (i) high turnaround latencies, and (ii) frequent sleeps and wake ups do not reduce power consumption. To fix both problems simultaneously, we recognize the following trade-off: We can let some threads sleep for long periods, while the rest coordinate with busy waiting. If the communication is mostly done via busy waiting, we almost remove the `futex` wake-up calls from the critical path. Additionally, we let threads sleep for long periods, a requirement for reducing power consumption in sleeping.

[5] Estimated as the required delay between sleep and wake-up calls for the wake-up calls to almost always find the other thread sleeping.

[6] The wake-up call latency is directly measured in our microbenchmark, while the turnaround time is estimated as the duration of the sleep call, reduced by the delay between the sleep and wake-up calls.

[7] When the core is constantly active due to multiprogramming, the turnaround latency only includes the scheduling delays.

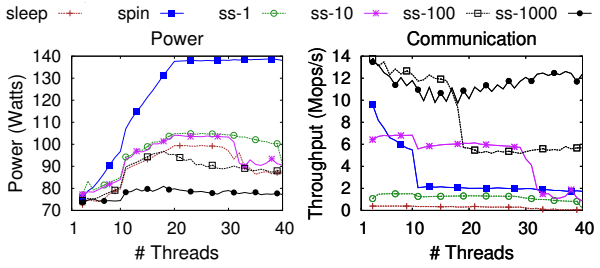


Figure 7: Power and communication throughput of sleeping, spinning, and spin-then-sleep for various Ts.[8]

This optimization comes at the expense of fairness. The longer a thread sleeps while some others progress, the more unfair the lock becomes. We experiment with the extreme case where only two threads communicate via busy waiting, while the rest sleep. Each active thread has a "quota" T of busy-waiting repetitions, after which it wakes up another thread to take its turn. Figure 7 shows the power and the communication rate (similar to a lock handover) of sleeping, busy waiting, and spin-then-sleep (*ss*-T) with various Ts on a single `futex`. T is the ratio of busy-waiting over `futex` handovers.

Figure 7 clearly shows that the more unfair an execution (i.e., for large Ts), the better the energy efficiency. First, larger values of T result in lower power, because the sleep and wake-up `futex` calls become infrequent, hence the sleeping threads sleep for a long duration. For example, on 10 threads with $T = 1000$, threads sleep for about 2M cycles. In comparison, with only sleeping, the sleep duration is less than 90000 cycles. Second, spin handovers face minimal contention, as only two threads attempt to "acquire" the cache line. Consequently, because most handovers (99.9%) happen with spinning, the latency is very low, resulting in high throughput.

Implications. Frequent `futex` calls will hurt the energy efficiency of a lock. A way around this problem is to reduce lock fairness in the face of high contention, by letting only a few threads use the lock as a spinlock, while the remaining threads are asleep.

5 Energy Efficiency of Locks

We evaluate the behavior of various locks in terms of energy efficiency and throughput, heavily relying on the results of §4. We first introduce MUTEXEE, an optimized version of MUTEX.

5.1 MUTEXEE: An Optimized MUTEX Lock

In §4, we analyze the overhead of `futex` calls. Additionally, we show how we can trade fairness for energy efficiency. MUTEX does not explicitly take these trade-offs into account, although it is an unfair lock.

[8]The performance collapse of *spin* is due to contention, while of *ss*-10 and *ss*-100 due to the high idle-to-active switching costs (see Figure 6).

	MUTEX	MUTEXEE
lock	for up to ∼ **1000 cycles** spin with `pause`	for up to ∼ **8000 cycles** spin with `mfence`
	if still busy, sleep with `futex`	
unlock	release in user space (`lock->locked = 0`)	**wait in user space**
	wake up a thread with `futex`	

Table 1: Differences between MUTEX and MUTEXEE.

In particular, MUTEX by default attempts to acquire the lock once before employing `futex`. MUTEX can be configured (with the `PTHREAD_MUTEX_ADAPTIVE_NP` initialization attribute) to perform up to 100 acquire attempts before sleeping with `futex`.[9] Still, threads spin up to a few hundred cycles on the lock before sleeping with `futex` (the exact duration depends on the contention on the cache line of the lock). This behavior can result in very poor performance for critical sections of up to 4000 cycles. In brief, threads are put to sleep, although the queuing time behind the lock is less than the `futex`-sleep latency. Additionally, to release a lock, MUTEX first sets the lock to "free" in user space and then wakes up one sleeping thread (if any). However, a third concurrent thread can acquire the lock before the newly awaken thread T_{aw} is ready to execute. T_{aw} will then find the lock occupied and sleep again, thus wasting energy, creating unnecessary contention, and breaking lock fairness.

To fix these two shortcomings, we design an optimized version of MUTEX, called MUTEXEE. Table 1 details how MUTEXEE differs from the traditional MUTEX. The "wait in user space" step of unlock requires further explanation. MUTEXEE, after releasing the lock in user space, but before invoking `futex`, waits for a short period to detect whether the lock is acquired by another thread in user space. In such case, the unlock operation returns without invoking `futex`. The waiting duration must be proportional to the maximum coherence latency of the processor (e.g., 384 cycles on Xeon).

Moreover, MUTEXEE operates in one of two modes: (i) *spin*, with ∼ 8000 cycles of spinning in the lock function and ∼ 384 in unlock, and (ii) *mutex*, with ∼ 256 cycles in lock and ∼ 128 in unlock (used to avoid useless spinning). MUTEXEE keeps track of statistics regarding how many handovers occur with busy waiting and with `futex`. Based on those statistics, MUTEXEE periodically decides on which mode to operate in: If the `futex`-to-busy-waiting handovers ratio is high (>30%), MUTEXEE uses mutex, otherwise it remains in spin mode.

[9]For brevity, in our graphs we show the default MUTEX configuration (i.e., without `PTHREAD_MUTEX_ADAPTIVE_NP`). We choose the default MUTEX version because: (i) it is the default in our systems (§6), and (ii) we thoroughly compare the two versions and conclude that for most configurations MUTEX is slightly faster without the adaptive attribute.

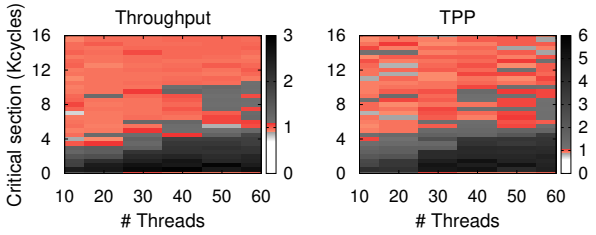


Figure 8: Throughput and TTP ratios of MUTEXEE over MUTEX on various configurations with a single lock.

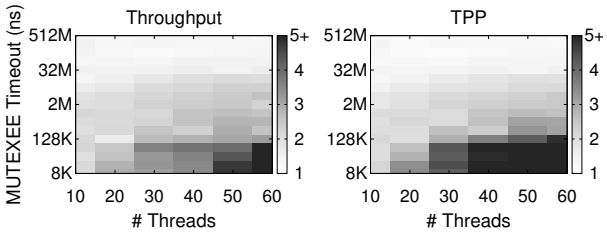


Figure 10: Throughput and TTP ratios of MUTEXEE without over with timeouts depending on the timeout.

Our design sensitivity analysis for MUTEXEE (not shown in the graphs) highlights three main points. First, spinning for more than 4000 cycles is crucial for throughput: MUTEXEE with 500 cycles spin behaves similarly to MUTEX. Second, the "wait in user space" functionality is crucial for power consumption (and improves throughput): If we remove it, MUTEXEE consumes similar power to MUTEX. Finally, the spin and mutex modes of MUTEXEE can save power on lengthy critical sections.

Fine-tuning MUTEXEE. The default configuration parameters of MUTEXEE should be suitable for most x86 processors. Still, these parameters are based on the latencies of the various events that happen in a `futex`-based lock, such as the latency of sleeping or waking up. Accordingly, in order to allow developers to fine-tune MUTEXEE for a platform, we provide a script which runs the necessary microbenchmarks and reports the configuration parameters that can be used for that platform.

Comparing MUTEXEE to MUTEX. Figure 8 depicts the ratios of throughput and energy efficiency of MUTEXEE over MUTEX on various configurations on a single lock. MUTEXEE indeed fixes the problematic behavior of MUTEX for critical sections of up to 4000 cycles. While MUTEX continuously puts threads to sleep and wakes them up shortly after, MUTEXEE lets the threads sleep for larger periods and keeps most lock handovers `futex` free. Of course, the latter behavior of MUTEXEE results in lower fairness as shown in Figure 9. Up to 4000 cycles, MUTEXEE achieves much lower 95th percentile latencies than MUTEX, because most lock handovers are fast with busy waiting. However, the price of this behavior is a few extremely high latencies as shown in the 99.99th percentile graph. These values are caused by the long-sleeping threads and represent the trade-off between lock fairness and energy efficiency. As the critical section size increases, the behavior of the two locks converges: Both locks are highly unfair.

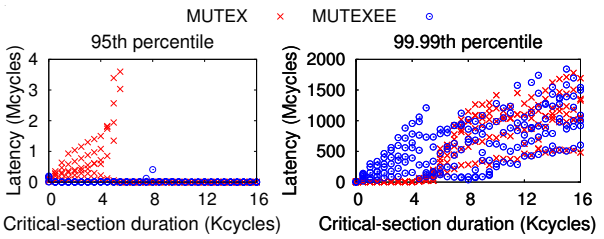


Figure 9: 95th and 99.99th percentile latency of a single MUTEX and MUTEXEE on various configurations.

Reducing MUTEXEE's Tail Latencies. MUTEXEE purposefully tries to reduce the number of `futex` invocations by handing the lock over in user space whenever possible. Therefore, it might let some threads sleep while the rest keep the lock busy, resulting in high tail latencies. A straightforward way to limit the tail latencies of MUTEXEE, so that threads are not allowed to remain "indefinitely" asleep, is to use a timeout for the `futex` sleep call. Once a thread is woken up due to a timeout, the thread spins until it acquires the lock, without the possibility to sleep again.[10] Controlling this timeout essentially controls the maximum latency of the lock (given that the sleep duration is significantly larger than the critical sections protected by that lock).

Figure 10 depicts the relative performance of MUTEXEE without over with timeouts for a single lock with 2000 cycles critical sections. For an 8 μs timeout, MUTEXEE delivers up to 14x lower throughput and 24x lower TPP than without timeouts. In general, for timeouts shorter than 16-32 ms, both throughput and TPP suffer, representing the clear trade-off between fairness and performance. For example, with 20 threads, MUTEXEE with a 4 ms timeout compares to the rest as follows:

Lock	Throughput Kacq/s	TPP Kacq/Joule	Max Latency Mcycles
MUTEX	317	4.0	2.0
MUTEXEE	855	10.9	206.5
MUTEXEE timeout	474	6.5	12.0

Depending on the application, the developer can decide whether to use timeouts and choose the timeout duration for MUTEXEE. For brevity, in the rest of the paper, we use MUTEXEE without timeouts. As we show in §6, we do not observe significant tail-latency increases due to MUTEXEE in real systems.

[10] Of course, one can design more elaborate variants of this protocol.

	MUTEX	TAS	TTAS	TICKET	MCS	MUTEXEE
Throughput	11.88	16.88	16.98	16.97	12.04	13.32
TPP	174.31	248.14	249.41	249.24	176.72	195.48

Table 2: Single-threaded lock throughput and TPP.

5.2 Evaluating Lock Algorithms

We evaluate various lock algorithms under different contention levels in terms of throughput and TPP.

Uncontested Locking. It is common in systems that a lock is mostly used by a single thread and both the acquire and the release operations are almost always uncontested. Table 2 includes the throughput (Macq/s) and the TPP (Kacq/Joule) of various lock algorithms when a thread continuously acquires and releases a single lock. We use short critical sections of 100 cycles.

The trends in throughput and TPP are identical as there is no contention. The locks perform inversely to their complexity. The simple spinlocks (TAS, TTAS, and TICKET) acquire and release the lock with just a few instructions. MUTEX performs several sanity checks and also has to handle the case of some threads sleeping when a lock is released. MUTEXEE is also more complex than simple spinlocks due to its periodic adaptation. The queue-based lock, MCS, is even more complex, because threads must find and access per-thread queue nodes.

Contention–Single (Global) Lock. We experiment with a single lock accessed by a varying number of threads. This experiment captures the behavior of highly-contended coarse-grained locks. We use a fixed critical section of 1000 cycles.

Figure 11 contains the throughput and the TPP results. On 40 threads, MUTEX delivers 73% lower TPP than TICKET: 63% less throughput and 5.8% more power. The throughput difference is due to (i) the global spinning of MUTEX, and (ii) the `futex` calls, even if they are infrequent. The power difference is mainly because of the pausing technique. MUTEX spins with `pause`, while TICKET uses a memory barrier. With `pause` instead of a barrier, TICKET consumes 4 Watts more.

Moreover, MUTEXEE maintains the contention levels and the frequency of `futex` calls low, regardless of the number of threads. This results in stable throughput and

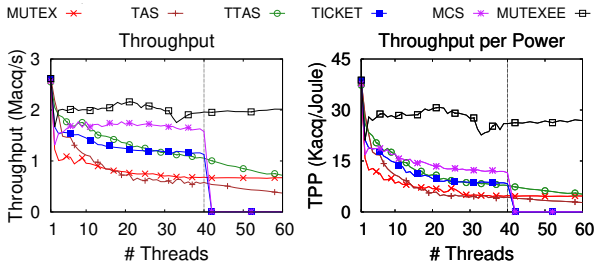


Figure 11: Using a single (global) lock.

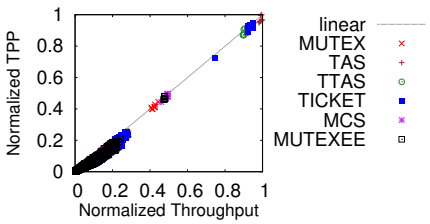


Figure 12: Correlation of throughput with energy efficiency (TPP) on various contention levels.

TPP because neither contention, nor the number of active hardware contexts increases with the number of threads. This behavior comes at the expense of high tail latency: On 40 threads, MUTEXEE has an 80x higher 99.9th percentile latency than MUTEX.

Regarding spinlocks, TAS is the worst in this workload. This behavior is due to the stress on the lock, which makes the release of TAS very expensive. Moreover, for up to 40 threads, the queue-based lock (MCS) delivers the best throughput and TPP. Queue-based locks are designed to avoid the burst of requests on a single cache line when the lock is released. On more than 40 threads, fairness shows its teeth. As Xeon has 40 hardware threads, there is oversubscription of threads to cores. TICKET and MCS, the two fair locks, suffer the most: If the thread that is the next to acquire the lock is not scheduled, the lock remains free until that thread is scheduled.

Finally, throughput and TPP are directly correlated: The higher the throughput, the higher the energy efficiency. Still, MUTEXEE delivers higher TPP by achieving both better throughput and lower power than the rest.

Variable Contention. Figure 12 plots the correlation of throughput with TPP on a diverse set of configurations. We vary the number of threads from 1 to 16, the size of critical section from 0 to 8000 cycles, and the number of locks from 1 to 512. At every iteration within a configuration, each thread selects one of the locks at random. The results are normalized to the overall maximum throughput and TPP, respectively.

Most data points fall on, or very close to, the linear line. In other words, most executions have almost one-to-one correlation of throughput with TPP. The bottom-left cluster of values represents highly-contended points. On high contention, there is a trend below the linear line, which represents executions where throughput is relatively higher compared to energy efficiency. These results are expected, as on very high contention sleeping can save power compared to busy waiting, but still, busy waiting might result in higher throughput.

If we zoom into the per-configuration best throughput and TPP, the correlation of the two is even more profound. On 85% of the 2084 configurations, the lock with the best throughput achieves the best energy efficiency

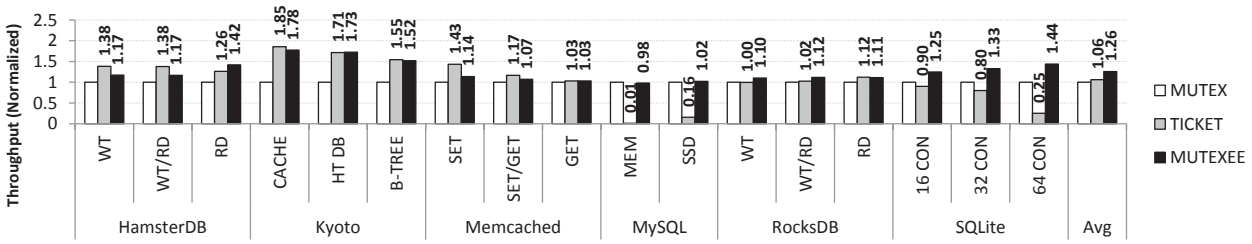


Figure 13: Normalized (to MUTEX) throughput of various systems with different locks. (Higher is better)

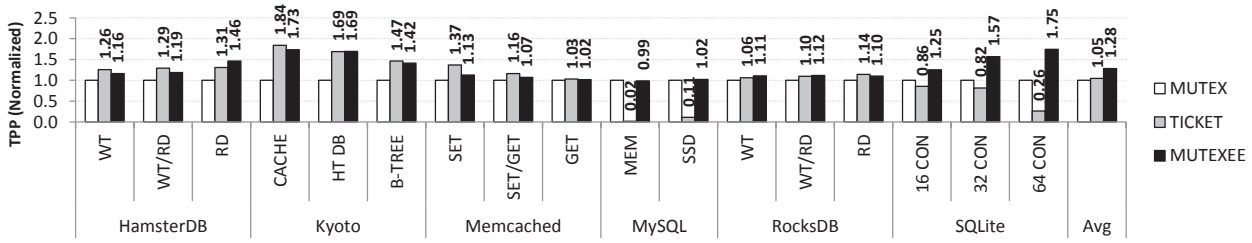


Figure 14: Normalized (to MUTEX) energy efficiency (TPP) of various systems with different locks. (Higher is better)

as well. On the remaining 15%, the highest throughput is on average 8% better than the throughput of the highest TPP lock, while the highest TPP is 5% better than the TPP of the highest throughput lock.

Finally, MUTEXEE delivers much higher throughput and TPP than MUTEX; on average, 25% and 32% higher throughput and TPP, respectively. MUTEX is better than MUTEXEE in just 4% of the configurations (by 9% on average, both in terms of throughput and TPP).

5.3 Implications

The POLY conjecture states that energy efficiency and throughput go hand in hand in locks. Our evaluation of POLY with six state-of-the-art locks on various contention levels shows that, with a few exceptions, POLY is indeed valid. The exceptions to POLY are high contention scenarios, where sleeping is able to reduce power, but still results in slightly lower throughput than busy waiting on the contended locks.

For low contention levels, energy efficiency depends only on throughput, as there are no opportunities for saving energy. In these scenarios, even infrequent `futex` calls reduce both throughput and energy efficiency.

For high contention, sleeping can reduce power consumption. However, the frequent `futex` calls of MUTEX hinder the potential energy-efficiency benefits due to throughput degradation. MUTEXEE is able to reduce the frequency of `futex` calls either by avoiding the ones that are purposeless, or by reducing fairness. MUTEXEE achieves both higher throughput and lower power than spinlocks or MUTEX for high contention levels.

6 Energy Efficiency of Lock-based Systems

We modify the locks of various concurrent systems to improve their energy efficiency. We choose the set of systems so that they use the pthread library in diverse ways, such as using mutexes or reader-writer locks, building on top of mutexes, or relying on conditionals. Note that we do not modify anything else other than the pthread locks and conditionals in these systems.

Table 3 contains the description and the different configurations of the six systems that we evaluate. All benchmarks use a dataset size of approximately 10 GB (in memory), except for the MySQL SSD configuration that uses 100 GB. We set the number of threads for each system according to its throughput scalability.

6.1 Results

Figures 13-14 show the throughput and the energy efficiency (TPP) of the target systems with different locks.

HamsterDB [3] Version: 2.1.7 # Threads: 4	An embedded key-value store. We run three tests with random reads and writes, varying the read-to-write ratio from 10% (WT), 50% (WT/RD), to 90% (RD).
Kyoto [7] Version: 1.2.76 # Threads: 4	An embedded NoSQL store. We stress Kyoto with a mix of operations for three database versions (CACHE, HT DB, B-TREE).
Memcached [8] Version: 1.4.22 # Threads: 8	An in-memory cache. We evaluate Memcached using a Twitter-like workload [40]. We vary the get-to-set ratio from 10% (WT), 50% (WT/RD), to 90% (RD). The server and the clients run on separate sockets.
MySQL [9] Version: 5.6.19	An RDBMS. We use Facebook's LinkBench and tuning guidelines [2] for an in-memory (MEM) and an SSD-drive (SSD) configurations.
RocksDB [10] Version: 3.3.0 # Threads: 12	A persistent embedded store. We use the benchmark suite and guidelines of Facebook for an in-memory configuration [11]. We run 3 tests with random reads and writes, varying the read-to-write ratio from 10% (WT), 50% (WT/RD), to 90% (RD).
SQLite [12] Version: 3.8.5	A relational DB engine. We use TPC-C with 100 warehouses varying the number of concurrent connections (i.e., 8, 32, and 64).

Table 3: Software systems and configurations.

For brevity, we show results with MUTEX, TICKET, and MUTEXEE. The remaining local-spinning locks are similar to TICKET (TAS is less efficient–see §5).

Throughput and Energy Efficiency. In 16 out of the 17 experiments, avoiding the overheads of MUTEX improves energy efficiency from 2% to 184%. On average, changing MUTEX for either TICKET or MUTEXEE improves throughput by 31% and TPP by 33%. The results include three distinct trends.

First, in some systems/configurations (i.e., Memcached and HamsterDB) sleeping can "kill" throughput. For instance, on the SET workload on Memcached, MUTEXEE allows for a few sleep invocations, resulting in lower throughput than TICKET.

Second, in some systems/configurations (i.e., MySQL and RocksDB) MUTEX is less of a problem. Both of these systems build more complex synchronization patterns on top of MUTEX. MySQL handles most low-level synchronization with customly-designed locks. Similarly, RocksDB employs a write queue where threads enqueue their operations and mostly relies on a conditional variable. Therefore, altering MUTEX with another algorithm does not make a big difference.

Finally, in MySQL and SQLite sleeping is necessary. Both these systems oversubscribe threads to cores, thus spinlocks, such as TICKET, result in very low throughput. A spinning thread can occupy the context of a thread that could do useful work. Additionally, on the SSD, TICKET consumes 40% more power than the other two, as it keeps all cores active. The fairness of TICKET exacerbates the problems of busy waiting in the presence of thread oversubscription: TTAS (not shown in the graph) has roughly 6x higher throughput than TICKET, but it is still much slower than MUTEX and MUTEXEE.

Overall, in five out of the six systems, the energy-efficiency improvements are mostly driven by the increased throughput. SQLite is the only system where the lock plays a significant role in terms of both throughput and power consumption. With MUTEXEE, SQLite consumes 15% and 18% less power than with MUTEX with 32 and 64 connections, respectively.

Tail Latency. MUTEXEE can become more unfair than MUTEX (see §5). Figure 15 includes the QoS of four systems in terms of tail latency. For most configurations, the results are intuitive: Better throughput comes with a lower tail latency. However, there are a few configurations that are worth analyzing.

First, MUTEXEE's unfairness appears in the RD configuration of HamsterDB, resulting in almost 20x higher tail latency than MUTEX, but also in 46% higher TPP. Second, TICKET has high tail latencies on all oversubscribed executions as a result of low performance.

Finally, MUTEXEE on SQLite achieves better throughput and lower power than MUTEX, without increasing tail latencies. TPC-C transactions on SQLite have latencies in the scale of tens of ms. Each transaction consists of multiple accesses to shared data protected by various locks. MUTEXEE does indeed increase the tail latency of individual locks, but these latencies are in the scale of hundreds of μs and do not appear in the transaction latencies. However, this low-level unfairness brings huge contention reductions. For instance, on 64 CON, the SQLite server with MUTEX puts threads to sleep for 472 μs on average, compared to 913 μs with MUTEXEE. The result is that with MUTEX, SQLite spends more than 40% of the CPU time on the `_raw_spin_lock` function of the kernel due to contention on `futex` calls. In contrast, MUTEXEE spends just 4% of the time on kernel locks, and 21% on the user-space lock functions.

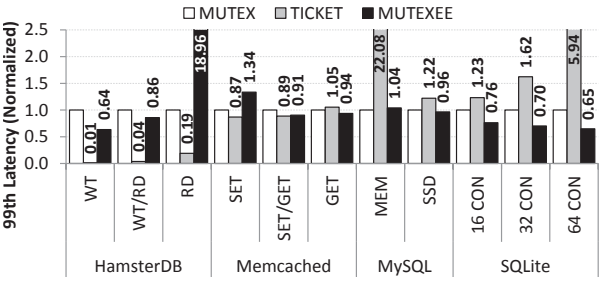


Figure 15: Normalized (to MUTEX) tail latency of various systems with different locks. (Lower is better)

Implications. Changing MUTEX in six modern systems results in 33% higher energy efficiency, driven by a 31% increase in throughput on average. Clearly, the POLY conjecture (i.e., throughput and energy efficiency go hand in hand in locks) holds in software systems and implies that we can continue business as usual: To optimize a system for energy efficiency, we can still optimize the system's locks for throughput.

Additionally, we show that MUTEX locks must be redesigned to take the latency overheads of `futex` calls into account. MUTEXEE, our optimized implementation of MUTEX, achieves 26% higher throughput and 28% better energy efficiency than MUTEX. Furthermore, the unfairness of MUTEXEE might not be a major issue in real systems: MUTEXEE can lead to high tail latencies only under extreme contention scenarios, that must be avoided in well engineered systems.

In conclusion, we see that optimizing lock-based synchronization is a good candidate for improving the energy efficiency of real systems. We can modify the locks with minimal effort, without affecting the behavior of other system components, and, more importantly, without degrading throughput.

7 Related Work

Lock-based Synchronization. Lock-based synchronization has been thoroughly analyzed in the past. For instance, many studies [13, 15, 34, 42, 46] point out scalability problems due to excessive coherence with traditional spinlocks and propose alternatives, such as hierarchical spin- and queue-based locks [34, 43, 54]. Prior work [20, 25] sacrifices short-term fairness for performance, however, it does not consider sleeping or energy efficiency. Similarly to MUTEXEE, Solaris' mutex locks offer the option of "adaptive unlock," where the lock owner does not wake up any threads if the lock can be handed over in user space [60]. David et al. [24] analyze several locks on different platforms and conclude that scalability is mainly a property of the hardware. Moreshet et al. [47] share some preliminary results suggesting that transactional memory can be more energy efficient than locks. Wamhoff et al. [62] evaluate the overheads of using DVFS in locks and show how to improve performance by boosting the lock owner. Our work extends prior synchronization work with a complete study of the energy efficiency of lock-based synchronization.

Spin-then-sleep Trade-off. The spin-then-sleep strategy was first proposed by Ousterhout [49]. Various studies [19, 35, 39] analyze this trade-off and show that just spinning or sleeping is typically suboptimal. Franke et al. [29] recognize the need for fast user-space locking and describe the first implementation of `futex` in Linux. Johnson et al. [33] advocate for decoupling the lock-contention strategy from thread scheduling. At first glance, MUTEXEE might look similar to their load-control TP-MCS [31] lock (LC). However, the two have some notable differences. LC relies on a global view of the system for load control, while MUTEXEE performs per-lock load control. LC's global load control can result in "unlucky" locks having their few waiting threads sleep for at least 100 ms, although there is low lock contention–sleeping threads are not woken up by a lock release, but only because of a decrease in load or 100 ms timeout. Finally, in contrast to MUTEXEE, LC might waste energy, because on low system load, no thread is blocked, even if the waiting times are hundreds ms.

Energy Efficiency in Software Systems. There is a body of work that points out the importance of energy-efficient software. For instance, Linux has rules to manage frequency and voltage settings [50]. Further work proposes OS facilities for managing and estimating power [48, 55, 58, 59, 65, 66]. Other frameworks approximate loops and functions to reduce energy [16, 57]. Moreover, compiler-based [63, 64] and decoupled access-execute DVFS [37] frameworks trade off performance for energy. In servers, consolidation [18, 21] collocates workloads on a subset of servers, and fast transitioning between active-to-idle power states allows for low idle power [44, 45]. Psaroudakis et al. [53] achieve up to 4x energy-efficiency improvements in database analytical workloads, using hardware models for power-aware scheduling. Similarly, Tsirogiannis et al. [61] analyze a DB system and conclude that the most energy-efficient point is also the best performing one. Our POLY conjecture is a similar result for locks. Nevertheless, while they evaluate various DB configurations, we study the spin vs. sleep trade-off. To the best of our knowledge, this is the first paper to consider the energy trade-offs of synchronization on modern multi-cores.

8 Concluding Remarks

In this paper, we thoroughly analyzed the power/performance trade-offs in lock-based synchronization in order to improve the energy efficiency of systems. Our results support the POLY conjecture: Energy efficiency and throughput go hand in hand in lock algorithms. POLY has important software and hardware ramifications.

For software, POLY conveys the ability to improve the energy efficiency of systems in an simple and systematic way, without hindering throughput. We indeed improved the energy efficiency of six popular software systems by 33% on average, driven by a 31% increase in throughput, These improvements are mainly due to MUTEXEE, our redesigned version of pthread mutex lock, that builds on the results of our analysis.

For hardware, POLY highlights the lack of adequate tools for reducing the power consumption of synchronization, without significantly degrading throughput. We performed our analysis on two modern Intel platforms that are representative of a large portion of the processors used nowadays. We argue that our results apply in most multi-core processors, because without explicit hardware support for synchronization, the power behavior of both busy waiting and sleeping will be similar regardless of the underlying hardware.

Our analysis further points out potential hardware tools that could reduce the power of synchronization. In brief, a truly energy-friendly `pause`, fast per-core DVFS, and user-level `monitor/mwait` can make the difference. In fact, industry has already started heading towards these directions. Intel includes on-chip voltage regulators on the latest processors, but all the cores share the same frequency. Similarly, the recent Oracle SPARC M7 processor includes a variable-length pause instruction and user-level `monitor/mwait` [51].

Acknowledgments. We wish to thank our shepherd, Leonid Ryzhyk, and the anonymous reviewers for their fruitful comments. We also want to thank Dave Dice for his useful feedback on an early draft of this paper. This work has been partially supported by the European ERC Grant 339539–AOC.

References

[1] America's Data Centers Consuming and Wasting Growing Amounts of Energy. http://www.nrdc.org/energy/data-center-efficiency-assessment.asp.

[2] Facebook LinkBench Benchmark. https://github.com/facebook/linkbench.

[3] HamsterDB. http://hamsterdb.com.

[4] Intel 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[5] Intel Xeon Processor E5-1600/ E5-2600/E5-4600 Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-1-datasheet.pdf.

[6] Java CopyOnWriteArrayList Data Structure. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CopyOnWriteArrayList.html.

[7] Kyoto Cabinet. http://fallabs.com/kyotocabinet.

[8] Memcached. http://memcached.org.

[9] MySQL. http://www.mysql.com.

[10] RocksDB. http://rocksdb.org.

[11] RocksDB In-memory Workload Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/RocksDB-In-Memory-Workload-Performance-Benchmarks.

[12] SQLite. http://sqlite.org.

[13] AGARWAL, A., AND CHERIAN, M. Adaptive Backoff Synchronization Techniques. ISCA '89.

[14] ANASTOPOULOS, N., AND KOZIRIS, N. Facilitating Efficient Synchronization of Asymmetric Threads on Hyper-threaded Processors. IPDPS '08.

[15] ANDERSON, T. The Performance of Spin Lock Alternatives for Shared-memory Multiprocessors. *IEEE TPDS* (1990).

[16] BAEK, W., AND CHILIMBI, T. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. PLDI '10.

[17] BARROSO, L., AND HÖLZLE, U. The Case for Energy-proportional Computing. *IEEE Computer* (2007).

[18] BENINI, L., KANDEMIR, M., AND RAMANUJAM, J. *Compilers and Operating Systems for Low Power*. Kluwer Academic Publishers, 2003.

[19] BOGUSLAVSKY, L., HARZALLAH, K., KREINEN, A., SEVCIK, K., AND VAINSHTEIN, A. Optimal Strategies for Spinning and Blocking. *J. Parallel Distrib. Comput.* (1994).

[20] CALCIU, I., DICE, D., LEV, Y., LUCHANGCO, V., MARATHE, V. J., AND SHAVIT, N. NUMA-Aware Reader-Writer Locks. PPoPP '13.

[21] CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., AND DOYLE, R. Managing Energy and Server Resources in Hosting Centers. SOSP '01.

[22] CRAIG, T. Building FIFO and Priority-Queuing Spin Locks From Atomic Swap. Tech. rep., 1993.

[23] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. ASPLOS '15.

[24] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. SOSP '13.

[25] DICE, D., MARATHE, V., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. PPoPP '12.

[26] ESMAEILZADEH, H., BLEM, E., S. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark Silicon and the End of Multicore Scaling. ISCA '11.

[27] ESMAEILZADEH, H., SAMPSON, A., CEZE, L., AND BURGER, B. Architecture Support for Disciplined Approximate Programming. ASPLOS '12.

[28] FLEISCHMANN, M. LongRun Power Management. Tech. rep., 2001.

[29] FRANKE, H., RUSSELL, R., AND KIRKWOOD, M. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. OLS '02.

[30] HARDAVELLAS, N., FERDMAN, M., FALSAFI, B., AND AILAMAKI, A. Toward Dark Silicon in Servers. *IEEE Micro* (2011).

[31] HE, B., III, W. N. S., AND SCOTT, M. L. Preemption Adaptivity in Time-Published Queue-Based Spin Locks. HiPC '05.

[32] HUNT, N., SANDHU, P., AND CEZE, L. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures. INTERACT '11.

[33] JOHNSON, F., STOICA, R., AILAMAKI, A., AND MOWRY, T. Decoupling Contention Management from Scheduling. ASPLOS '10.

[34] KÄGI, A., BURGER, D., AND GOODMAN, J. R. Efficient Synchronization: Let Them Eat QOLB. ISCA '97.

[35] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical Studies of Competitive Spinning for a Shared-memory Multiprocessor. SOSP '91.

[36] KOOMEY, J. Growth in Data Center Electricity Use 2005 to 2010. Analytics Press Report.

[37] KOUKOS, K., BLACK-SCHAFFER, D., SPILIOPOULOS, V., AND KAXIRAS, S. Towards More Efficient Execution: A Decoupled Access-execute Approach. ICS '13.

[38] LI, H., BHUNIA, S., CHEN, Y., ROY, K., AND VIJAYKUMAR, T. DCG: Deterministic Clock-Gating for Low-Power Microprocessor Design. *IEEE TVLSI* (2004).

[39] LIM, B.-H., AND AGARWAL, A. Waiting Algorithms for Synchronization in Large-scale Multiprocessors. *ACM TOCS* (1993).

[40] LIM, K., MEISNER, D., SAIDI, A., RANGANATHAN, P., AND WENISCH, T. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. ISCA '13.

[41] LO, D., CHENG, L., GOVINDARAJU, R., BARROSO, A., AND KOZYRAKIS, C. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. ISCA '14.

[42] LOZI, J., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. ATC '12.

[43] LUCHANGCO, V., NUSSBAUM, D., AND SHAVIT, N. A Hierarchical CLH Queue Lock. ICPP '06.

[44] MEISNER, D., GOLD, B., AND WENISCH, T. PowerNap: Eliminating Server Idle Power. ASPLOS '09.

[45] MEISNER, D., AND WENISCH, T. DreamWeaver: Architectural Support for Deep Sleep. ASPLOS '12.

[46] MELLOR-CRUMMEY, J., AND SCOTT, M. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM TOCS* (1991).

[47] MORESHET, T., BAHAR, R. I., AND HERLIHY, M. Energy Implications of Multiprocessor Synchronization. SPAA '06.

[48] MUTHUKARUPPAN, T., PATHANIA, A., AND MITRA, T. Price Theory Based Power Management for Heterogeneous Multi-Cores. ASPLOS '14.

[49] OUSTERHOUT, J. Scheduling Techniques for Concurrent Systems. ICDCS '82.

[50] PALLIPADI, V., AND STARIKOVSKIY, A. The Ondemand Governor. OLS '06.

[51] PHILLIPS, S. M7: Next Generation SPARC. Hot Chips '14.

[52] POWELL, M., YANG, S., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. Gated-Vdd: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. ISLPED '00.

[53] PSAROUDAKIS, I., KISSINGER, T., POROBIC, D., ILSCHE, T., LIAROU, E., TÖZÜN, P., AILAMAKI, A., AND LEHNER, W. Dynamic Fine-Grained Scheduling for Energy-Efficient Main-Memory Queries. DaMoN '14.

[54] RADOVIC, Z., AND HAGERSTEN, E. Hierarchical Backoff Locks for Nonuniform Communication Architectures. HPCA '03.

[55] RIBIC, H., AND LIU, Y. Energy-Efficient Work-Stealing Language Runtimes. ASPLOS '14.

[56] ROTEM, E., NAVEH, A., MOFFIE, M., AND MENDELSON, A. Analysis of Thermal Monitor Features of the Intel Pentium M Processor. TACS Workshop at ISCA '04.

[57] SAMPSON, A., DIETL, W., FORTUNA, E., GNANAPRAGASAM, D., CEZE, L., AND GROSSMAN, D. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. PLDI '11.

[58] SHEN, K., SHRIRAMAN, A., DWARKADAS, S., ZHANG, X., AND CHEN, Z. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers. ASPLOS '13.

[59] SINGH, K., BHADAURIA, M., AND MCKEE, S. Real Time Power Estimation and Thread Scheduling via Performance Counters. *SIGARCH CAN* (2009).

[60] SUN MICROSYSTEMS. Multithreading in the Solaris Operating Environment. `http://home.mit.bme.hu/~meszaros/edu/oprendszerek/segedlet/unix/2_folyamatok_es_utemezes/solaris_multithread.pdf`.

[61] TSIROGIANNIS, D., HARIZOPOULOS, S., AND SHAH, M. Analyzing the Energy Efficiency of a Database Server. SIGMOD '10.

[62] WAMHOFF, J., DIESTELHORST, S., FETZER, C., MARLIER, P., FELBER, P., AND DICE, D. The TURBO Diaries: Application-controlled Frequency Scaling Explained. ATC '14.

[63] WU, Q., MARTONOSI, M., CLARK, D., REDDI, V., CONNORS, D., WU, Y., LEE, J., AND BROOKS, D. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. MICRO '05.

[64] XIE, F., MARTONOSI, M., AND MALIK, S. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. PLDI '03.

[65] XU, C., LIN, F., WANG, Y., AND ZHONG, L. Automated OS-Level Device Runtime Power Management. ASPLOS '15.

[66] ZHAI, Y., ZHANG, X., ERANIAN, S., TANG, L., AND MARS, J. Happy: Hyperthread-Aware Power Profiling Dynamically. ATC '14.

Greening The Video Transcoding Service With Low-Cost Hardware Transcoders[1]

Peng Liu[*], Jongwon Yoon[†], Lance Johnson[‡], and Suman Banerjee[*]

[*]University of Wisconsin-Madison, {pengliu,suman}@cs.wisc.edu
[†]Hanyang University, jongwon@hanyang.ac.kr
[‡]University of Minnesota, lmj@umn.edu

Abstract

Video transcoding plays a critical role in a video streaming service. Content owners and publishers need video transcoders to adapt their videos to different formats, bitrates, and qualities before streaming them to end users with the best quality of service. In this paper, we report our experience to develop and deploy VideoCoreCluster, a low-cost, highly efficient video transcoder cluster for live video streaming services. We implemented the video transcoder cluster with low-cost single board computers, specifically the Raspberry Pi Model B. The quality of the transcoded video delivered by our cluster is comparable with the best open source software-based video transcoder, and our video transcoders consume much less energy. We designed a scheduling algorithm based on priority and capacity so that the cluster manager can leverage the characteristics of adaptive bitrate video streaming technologies to provide a reliable and scalable service for the video streaming infrastructure. We have replaced the software-based transcoders for some TV channels in a live TV streaming service deployment on our university campus with this cluster.

1 Introduction

Video streaming service is one of the most popular Internet services in recent years. In particular, multimedia usage over HTTP accounts for an increasing portion of today's Internet traffic [47]. For instance, video traffic is expected to be 80 percent of all consumer Internet traffic in 2019, up from 64 percent in 2014 [5]. In order to provide high and robust quality of video streaming services to end users with various devices with diverse network connectivities, content owners and distributors need to encode the video to different formats, bitrates, and qualities. It is redundant to encode and store a source video to different variants for archiving purposes. The broad spectrum of varieties of bitrates, codecs and formats make it difficult for some video service providers to prepare all media content in advance. Therefore, video transcoding has been widely used for optimizing video data. For example, Netflix encodes a movie as many as 120 times before they stream the video to users [37]. Transcoders are also commonly used in the area of mobile device content adaptation, where a target device does not support the format or has a limited storage capacity and computational resource that mandate a reduced file size.

However, video transcoding is a very expensive process, requiring high computational power and resources. Thus, it is critical to have an energy efficient and low-cost video transcoding solution. In addition, video transcoders' performance is important to enhance the overall quality of video streaming services. Regarding the performance of transcoder, two metrics are important: quality and speed. Video quality with a given bitrate determines the amount of data needed to be transmitted in the network for a video. Transcoding speed determines the time to finish the transcoding. Thus, it is critical for live video streaming services. Other metrics, e.g., cost and power consumption, are also need to be considered in video transcoding system deployment.

Various video transcoding technologies are proposed and used, including cloud transcoding, software-based transcoding on local servers, and hardware transcoding with specialized processors. In this paper, we introduce VideoCoreCluster – a low-cost, energy efficient hardware-assisted video transcoder cluster to provide transcoding services for a live video streaming service. The cluster is composed of a manager and a number of cheap single board computers (Raspberry Pi Model B). We use the hardware video decoder and encoder modules embedded in the System on Chip (SoC) of a Raspberry Pi to facilitate video transcoding. With an optimized transcoding software implementation on the Raspberry Pi, each Raspberry Pi is able to transcode up to 3 Standard Definition (SD, 720x480) videos or 1 High

Definition (HD, 1280x720) and 1 SD videos in real time with very low power consumption. We also developed the cluster manager based on an IoT machine-to-machine protocol - MQTT [42] to coordinate the transcoding tasks and hardware transcoders in order to provide reliable transcoding service for video streaming services. Compared to software-based video transcoders, VideoCoreCluster has lower cost and higher energy efficiency.

Adaptive bitrate (ABR) video streaming over HTTP is a popular technology to provide robust quality of service to end users whose mobile devices have dynamic network connectivities. Multiple ABR technologies are used in the industry, but they share a similar idea. Media servers cut source videos into small segments and encode every segment to multiple variants with different bitrates. During playback, a video player selects the best variants for these segments base on the video player's performance, network connectivity, or user's preference. Streaming video over HTTP has many advantages [60], we chose it for our live video streaming service because it is easy to deploy a video player on web browsers of mobile devices with different operating systems, and we do not need to reconfigure middle-boxes (firewalls, NATs, etc.) in current campus network deployment for our service. We design VideoCoreCluster according to the requirements of ABR live video streaming so that it can provide robust transcoding service.

Contributions: The most important contribution of this paper is a practical implementation of real-time transcoding system based on energy efficient hardware-assisted video transcoders. Our contributions are multi-fold:

- We design and implement a cost-effective transcoding solution on a low-cost device, Raspberry Pi Model B.
- Our system is based on hardware video decoder and encoder, which is significantly more energy efficient than software-based transcoding system.
- We leverage characteristics of adaptive bitrate video streaming over HTTP to design a reliable and scalable system for live video streaming service. The system is easily deployable. We currently employ our VideoCoreCluster into an IP-based TV streaming service in the University of Wisconsin-Madison.

The paper is organized as follows. In Section 2, we introduce the background of our video transcoding system and the possible approaches to building it. Then we describe the architecture of VideoCoreCluster in Section 3. We evaluate the system in Section 4 and discuss the related work in Section 5. Section 6 concludes the paper.

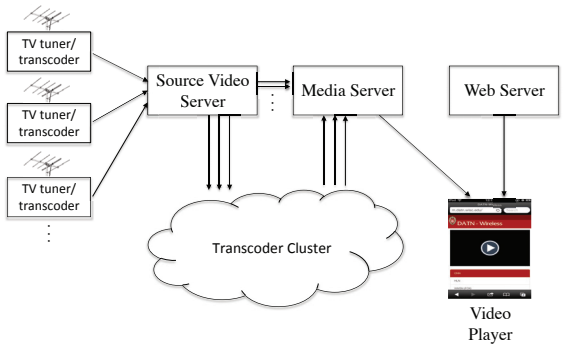


Figure 1: TV Streaming Service Architecture.

2 Background and Setup

We worked with the IT department of the University of Wisconsin-Madison to provide a free TV service for students on campus. Students can watch 27 (currently) TV channels with their mobile devices. Six of these channels are HD channels with a resolution 1280x720, 30fps. The other 21 channels are SD with resolutions up to 720x480, 30fps. In the most recent month (April 2016), there were more than 4000 view sessions and about total 480 watching hours. ABR video streaming techniques are used to provide high-quality video streaming service to mobile devices with diverse link capabilities and network dynamics. The system supports both Apple's HTTP Live Streaming (HLS) [13] and MPEG-DASH [60] to provide service to heterogeneous devices. Figure 1 shows the architecture of the system. Specifically, a TV frontend receives the TV signal and encodes the video stream into H.264 + AAC format, then the video stream is pushed to *source video server*. The *transcoder cluster* pulls videos from the *source video server* and pushes the transcoded results to *media server* in order to provide multiple variants of the video streams for every TV channel. The *source video server* and *media server* can be combined in deployment, so we will not discuss them separately in the following sections. The web server hosts web pages and video players for different web browsers.

2.1 Challenges of ABR Techniques on Live Video Streaming

Low latency is critical for a live video streaming service, in which a media server has to generate video data on-the-fly to provide continuous streaming service to the end user. A media server supporting ABR needs to guarantee all the variants of a video segment are available when a client requests one of them. Due to the strict requirement on low-latency transcoding, several optimization techniques (e.g., high throughput video transcoding,

multi-pass encoding for enhancing video quality) are not applicable for live streaming. Moreover, an index file containing a list of the available video variants should be generated in real-time, and it has to be updated on time and be consistent with the availability of these variants. We also need to make sure the variants of a video segment are generated synchronously to simplify the implementation of ABR algorithms on the video players. Furthermore, an efficient video transcoding system is desired and high reliability is required to provide 24/7 video streaming services to users without interruptions.

2.2 Video Transcoding

A video transcoder is composed of a video decoder and an encoder. Some researchers have discussed possible video transcoding designs, which mingle the modules of a video decoder and an encoder [40]. However, having separate video decoders and encoders provides more flexibility because we can easily have various decoder/encoder combinations to create different video transcoders for various purposes. In this work, we only discuss the video transcoder built by a separate video decoder and an encoder.

Most popular video codecs have well-defined standards. These standards strictly define the compliant bitstreams and decoder's behaviors. For many video coding techniques, including H.264 [46], different video decoder implementations are required to generate the identical video frames (output) with respect to the same input by their standards. This makes the selection of video decoder easy, and hence, hardware video decoders would be the best choice in most cases as long as it is available at low cost due to its high efficiency.

On the other hand, developers are free to design a video encoder's implementation as long as the generated bitstream can be decoded by the reference decoder implementation. Therefore, different encoder implementations can generate different bitstreams with various qualities for the same video frames. In this way, the interoperability is guaranteed while innovations on the encoder design are encouraged. As a result, we need to evaluate an encoder's performance on video quality in addition to the encoding speed when we select an encoder for a transcoder. Software video encoders are well known for their low efficiency. Hameed et al. [48] point out that application-specific integrated circuit (ASIC) implementation of the video encoder is 500 times more energy efficient than the software video encoder running on general purpose processors. They assume that both hardware and software implementations use the same algorithms for the corresponding procedures, e.g., motion estimation, intra-prediction, etc.. The high efficiency is only from the advantage of specialized hardware implementation. However, software video encoders are more flexible than hardware video encoders. It is much easier to try new algorithms on a software video encoder to improve video quality than a hardware video encoder. Video encoders on FPGA platform make a good trade-off on efficiency and flexibility, so they are also widely used in industry.

Different video applications have different requirements on video encoder. For mobile devices, the energy efficiency is crucial for battery life. Therefore, most of the SoCs for mobile phones include video decoder/encoder IPs [7, 36]. For broadcast applications, high video quality is desired while real-time encoding and flexibility are critical. Toward this, video encoders on the FPGA platform are widely used. For on-demand streaming applications, the low energy efficiency of software video encoders can be amortized by streaming one encoded result to many users. High latency is not a big concern because the service provider can encode video offline. Slow encoding speed can be overcome by launching a large number of instances in a cloud platform to run the video encoders in parallel to achieve very high throughput video encoding. The advantages of video quality and flexibility are the main reason that software video encoder is widely used to prepare video contents for on-demand streaming service. For instance, Netflix adapts the software-based transcoder because of its flexibility, after an unsuccessful deployment of a specialized hardware video transcoding system [23].

H.264 is a popular video standard widely used in diverse video applications. Since H.264 is the video coding standard that our system supports, we only discuss the available H.264 encoder implementations in this paper as shown in Table 1. The authors in [6] compared different H.264 encoder implementations, which includes software implementations (x264, DivX H.264, etc.), GPU-accelerated implementation (MainConcept CUDA), and hardware implementation (Intel QuickSync Video). Their conclusions include *(i)* x264 is one of the best codecs regarding video quality, and *(ii)* Intel QuickSync is the fastest encoder of those considered. Even though x264 is the best software H.264 encoder for our project, its low efficiency hinders its deployment. Given that we need to keep the system running 24/7 for continuous TV service in the campus network, it is not economical to rent cloud computing resource (e.g., Amazon EC2 instances) to execute the transcoding tasks. Building in-house transcoding system with highly efficient hardware video decoders and encoders is the best choice while keeping the cost low in our system.

We use the hardware H.264 decoder and encoder in a low-cost SoC – Broadcom BCM2835, which is the chip of a popular single board computer - Raspberry Pi Model B. VideoCoreCluster leverages its powerful GPU - VideoCore IV, which embeds hardware multimedia de-

Encoder Type	Explanations
Software	JM: The reference H.264 implementation from JVT[11]. It is widely used for research purpose and conformance test. It is too slow to be used in practical projects. x264: The most popular open source software H.264 encoder implementation. Compared to JM, it is about 50 times faster and provides bitrates within 5% of the JM reference encoder for the same PSNR[38, 57]. OpenH264: An open source H.264 implementation from Cisco[28]. It is optimized and the encoder runs much faster than JM, but it is slower and has fewer features than x264. Other proprietary implementations: MainConcept[17], Intel's IPP H.264 encoder[15], etc.
GPU-based	Three GPU vendors: Intel, NVIDIA, and AMD, all integrate hardware video codecs in their GPUs. They also provide GPU-accelerated video encoders, which leverage GPU's high throughput graphics engine to accelerate video encoding[27, 16]. For example, NVIDIA has two different versions of video encoder implementations: NVCUVENC and NVENC. NVCUVENC is a CUDA software-based implementation while NVENC is based on dedicated encoding hardware engine. NVCUVENC will not be available in the future because NVENC's improved performance and quality.
FPGA-based	Xilinx and its partners have professional solutions for broadcast applications[10]. They provide H.264 decoder and encoder IP for Xilinx's FPGA platforms. FPGA-based implementation is more flexible than ASIC implementation and more efficient than the software implementation. But it is more expensive than both of them.
Encoder IP in SoC	Many SoCs for mobile devices or other embedded devices have dedicated hardware decoders and encoders. For example, Qualcomm's chips for mobile phones [7]; Ambarella's chips for different video applications [2]; Broadcom's chip for TV set-top boxes.

Table 1: Different type of H.264 encoders

coders and encoders. The primary computation power of the cluster is from the VideoCore co-processors. The Raspberry Pi is a low cost (under $35) single board computer with very good software support [31, 33, 35]. Therefore, it is adequate to build a cost-effective video transcoder cluster.

3 VideoCoreCluster Architecture

The VideoCoreCluster is composed of a cluster manager and a number of transcoders. We use the MQTT protocol to transfer control signals between the cluster manager and the transcoders. MQTT is based on a publish/subscribe messaging pattern rather than a traditional client-server model, where a client communicates directly with a server. The cluster manager and the transcoders connect to an MQTT message broker. They exchange information by subscribing to topics and publishing messages to topics. The message payload is encoded with Google Protocol Buffer for minimal overhead [32]. RTMP [34] is used in the data path. Figure 2 shows the architecture of the VideoCoreCluster.

3.1 Media Server

The media server supports HLS, MPEG-DASH, and RTMP. HLS and DASH are for the video players, whereas RTMP is for the transcoders. HLS and DASH are two popular adaptive bitrate streaming standards to stream video over HTTP and they are widely supported by mobile operating systems and web browsers. RTMP is designed to transmit real-time multimedia data, and thus, it guarantees to transfer source video to the transcoders and the transcoded video to the media server with minimum latency. RTMP's session control features for media applications are used to implement the interactions between media server and transcoders in our configuration.

Figure 3 shows the responsibilities of the media server in our system. For every TV channel, there are multiple transcoded video streams (variants) with different bitrates and qualities. Each video stream is split to chunks with the same duration on the media server. It is important to align the boundaries of those chunks from different variants even different workers may not be exactly synchronized. To ensure synchronization, we set the Instantaneous Decoder Refresh (IDR) interval of a source video and the transcoded video to 2 seconds. Then the media server uses the IDR frame as the boundary to split the streams. It uses the timestamp of the IDR frame with 2 seconds (IDR interval) granularity to define the segment number. So that we can keep the chunks to be synchronized from a video player's view as long as the progress offset of different workers for the same source video is under 2 seconds (IDR interval). We can set the IDR interval to a larger value to obtain higher tolerance on transcoding speed variation. The index file for a channel has the information about all variants of the video. It is dynamically updated by the media server based on the

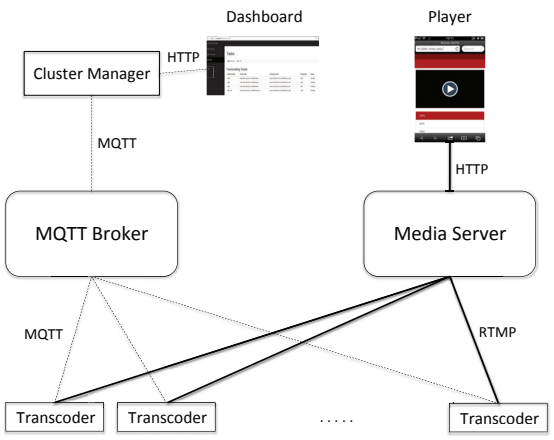


Figure 2: The components and connections between the components in the VideoCoreCluster. Every transcoder node maintains two separate network connections for the control flow and data flow respectively. Administrator monitors the status of VideoCoreCluster and updates transcoding tasks through the dashboard of the cluster manager.

availability of the video streams. One worker may temporarily fail to generate the corresponding stream, and the transcoder cluster can migrate the failed task from one worker to another within a second. RTMP specification requires the initial timestamp of a stream to be 0 [34]. To ensure all other RTMP streams for the same channel as the failed stream have the same timestamp, we need to reset their RTMP connections as well. And the media server can always generate a consistent index file so that we can guarantee the reliability of the video streaming service.

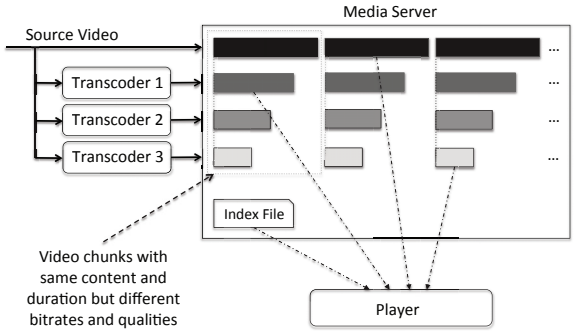


Figure 3: Overview of the Media Server's responsibilities: (i) Splitting video streams to chunks with the same duration (IDR interval). (ii) Refreshing the index file according to the status of the transcoded streams.

3.2 Transcoder Design

The Raspberry Pi Model B has a weak ARM CPU but a powerful GPU (VideoCore IV). Given that, our design strategy is to run simple tasks on the CPU, and offload compute-intensive tasks to the GPU. There are two types of processes in a transcoder: cluster agent and transcoding worker. There is one cluster agent on the board, and from 0 to 3 transcoder workers depending on the transcoding task's requirement on the computing resource. Figure 4 shows the relationship between these two types of processes.

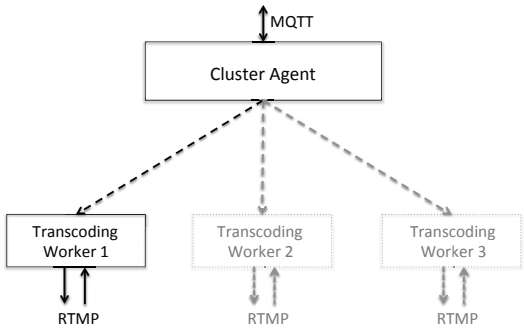


Figure 4: Two types of processes on a transcoder. A transcoding worker is the child process of the cluster agent process.

A cluster agent executes on the ARM processor only. It has two responsibilities: (i) Reporting the status of the board and workers to the cluster manager. (ii) Accepting commands from the cluster manager and launching or killing transcoding worker processes. A cluster agent will report the available resource to the cluster manager when it registers to the cluster manager. The cluster manager dispatches transcoding tasks to a transcoder based on the available resources of the transcoder. A cluster agent maintains an MQTT connection to the MQTT message broker by sending keep-alive packets if no information flows between the transcoder and the MQTT message broker for a predefined interval. If a cluster agent is disconnected from the broker or the cluster manager is offline, the transcoder will stop all transcoding workers. The cluster manager can revoke a transcoding task from a transcoder if it wants to assign the task to another transcoder. Failed transcoding tasks will be reported to the cluster manager, which can assign them to other transcoders.

A transcoding worker is a process to transcode a video stream. It only transcodes video data but passes audio data. Video transcoding tasks primarily execute on the VideoCore IV co-processor. The ARM processor is responsible for executing the networking protocol stack, video streams demuxing/muxing, and audio data pass

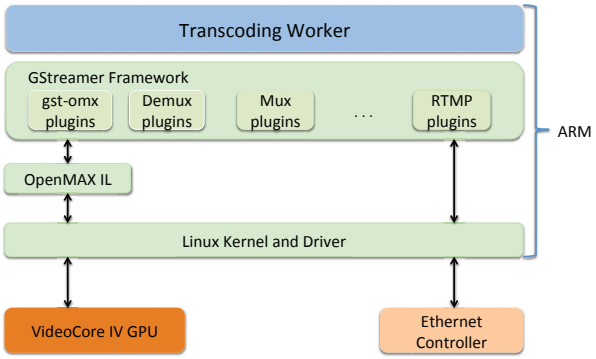


Figure 5: Software architecture of a transcoding worker. Compute-intensive video transcoding task executes on VideoCore IV GPU, whereas ARM processor is responsible for coordination and data parsing/movement.

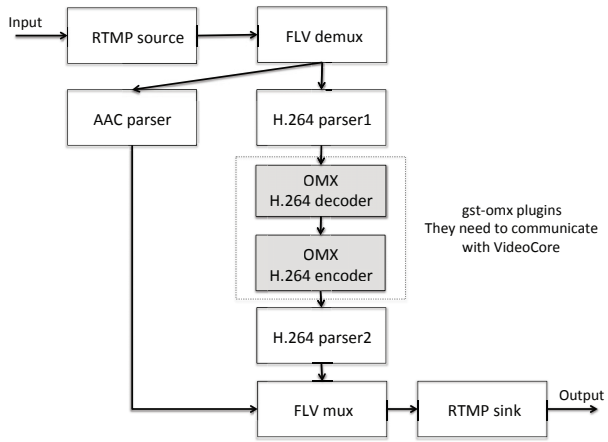


Figure 6: GStreamer pipeline of the transcoding worker process. Some trivial plugins, e.g., buffering, synchronization, etc. are not shown here.

through. In BCM2835, an application layer software can access the hardware video decoder and encoder through the OpenMAX IL interfaces [31, 29]. Rather than calling the OpenMAX IL interfaces directly, we built the program with the GStreamer framework [9]. GStreamer is an open source multimedia framework widely used to build media processing applications. It has a well-designed filter (plugin) system. The gst-omx is the GStreamer OpenMAX IL wrapper plugin that we use to access the hardware video decoder and encoder resources. Figure 5 shows the software architecture of a transcoding worker.

A transcoding worker creates a pipeline of GStreamer plugins. Video data are processed by the plugins one by one, sequentially. Figure 6 shows the structure of a pipeline. All plugins except H.264 decoder and H.264 encoder plugins fully execute on the ARM processor.

The default behavior of a GStreamer plugin can be summarized as 3 steps: (i) Read data from the source pad. (ii) Process the data. (iii) Write data to the sink pad. The GStreamer pipeline moves data and signals between the connected source pad and sink pad. Plugins work separately and process the data sequentially. This means both the H.264 decoder and H.264 encoder need to move a large amount of data back and forth between the memories for the ARM processor and the VideoCore IV GPU, which wastes CPU cycles. We modified the gst-omx implementation to enable hardware tunneling between the decoder and encoder, which significantly reduce the CPU load [8]. Without the hardware tunneling, a transcoder worker cannot support real-time transcoding of a 1280x720, 30fps video. Whereas after we enable it, a transcoder worker can simultaneously transcode one 1280x720 video and one 720x480 video in real-time. Figure 7 illustrates the data movement in the pipeline without hardware tunneling. Figure 8 illustrates the data movement in the pipeline with hardware tunneling.

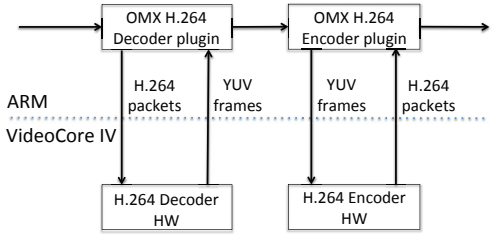


Figure 7: The decoder and encoder plugins work independently. YUV frames need to be moved from the VideoCore IV memory to the ARM memory by the decoder plugin, then they need to be moved from the ARM memory to the VideoCore IV memory by the encoder plugin.

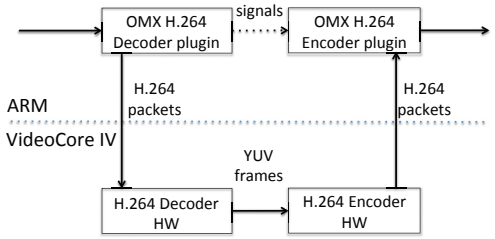


Figure 8: A hardware tunnel is created between the decoder and encoder. Plugins do not need to touch the YUV frames. Therefore, the workload of the ARM processor is reduced.

3.3 Cluster Manager Design

The cluster manager maintains a task pool and a transcoder pool. Its major responsibilities are assigning the transcoding tasks to the transcoders, and migrating the failed tasks from one transcoder to another. Both transcoding tasks and transcoders have priorities, which need to be considered when the cluster manager schedules the tasks. The cluster manager maintains a series of lists of tasks with different priorities. Every transcoding task has the following information: { ID, channel name, command, bitrate, resource, priority, state }. The resource is an integer representing the required computational resource for the task. Its value depends on the source video type (SD or HD) and target bitrate. Each task has four possible states: idle, assigning, revoking, and running. The cluster manager employs an event-driven design. When anything happens to the tasks or transcoders, e.g., task failure, a new worker joining, a worker leaving, etc., the cluster manager will check whether rescheduling is necessary and do so if it is.

When a new worker sends a register message to the cluster manager, the cluster manager will add a record to the worker list and keep tracking its status with a predefined interval. Transcoders will send the status of themselves and tasks running on them to the cluster manager periodically. The transcoder status includes CPU load, VideoCore IV load, temperature, and free memory. MQTTs *last will message* mechanism is used to implement the online status tracking of the transcoders. The cluster manager also embeds a web server to provide a dashboard for administrators to monitor the cluster's status and change the configurations. We have three design goals for the cluster manager: scalability, reliability, and elasticity.

Scalability: Scalability is ensured by an event-driven design and minimum control flow information. The cluster manager only maintains critical information about the transcoders. The information that frequently exchanges between the cluster manager and transcoders is only about the status. Media servers manage the source videos and transcoded videos. We can replicate the media server if its workload is too high. The separation of data flow and control flow ensures the cluster manager's scalability.

Reliability: We ensure the reliability of the system by real-time status monitoring coupled with low latency scheduling to migrate the failed tasks to working transcoders. The media server updates the index file on-the-fly to consistently list correctly transcoded video streams. A temporary failure will not affect the video players.

Elasticity: We define priorities for tasks and transcoders. An important characteristic of adaptive bitrate video streaming is that every video program has multiple versions of encoded videos. The more variants of a video available, the more flexible the players can optimize the user experience. Our system leverages that characteristic to implement an elastic transcoding service. When the available transcoders have more resources to run all the transcoding tasks, all the tasks will be assigned to transcoders. But if not, only the high priority tasks will be scheduled and executed. We can easily extend the transcoder cluster by adding more transcoders with this elastic design to support more TV channels.

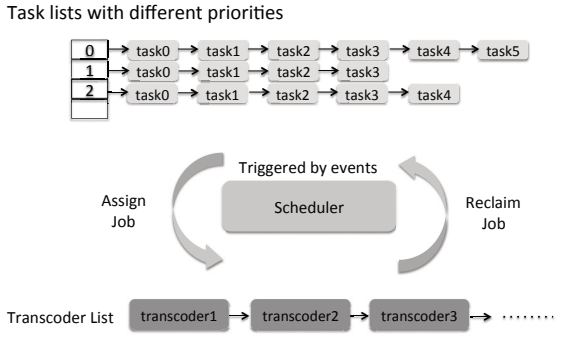


Figure 9: Overview of the cluster manager. The scheduler is an event-driven design.

Figure 9 shows the internal data structures of the cluster manager. The scheduler makes decisions based on the capacities of the transcoders and the resource requirements of the transcoding tasks. The capacity and resource requirement are both positive integers. Their values determine what kind of tasks and how many tasks can run on a transcoder in real-time.

Table 2 presents several task examples. A Raspberry Pi Model B's capacity is 20, so it can run one HD transcoding task, e.g., 4 or 5, or two SD transcoding tasks (1 and 2), or three SD transcoding tasks (2, 3, and 7), or one HD transcoding task and 1 SD transcoding task (3 and 6). When a new transcoder registers to the cluster manager, the idle task in the highest priority task list will be assigned to it. When a task fails and returns to the cluster manager, the task manager will try to assign it to another transcoder. If the cluster manager can find a transcoder that has enough capacity for it, it will assign the failed task to that transcoder. If not, the cluster manager will try to revoke tasks with lower priorities from a transcoder and then assign this task to that transcoder. If the cluster manager can not find a running task which has lower priority than the failed task, the cluster manager will not do anything. As discussed in section 3.1, if we successfully reschedule a failed task, we need to send messages to transcoders to reset the RTMP connections of those streams corresponding to the same channel as

ID	Channel	Command	Resource	Priority
1	A	transcode rtmp://192.168.1.172:1935/live/A_src rtmp://192.168.1.172:1935/live/A_800k 800	10	0
2	A	transcode rtmp://192.168.1.172:1935/live/A_src rtmp://192.168.1.172:1935/live/A_600k 600	8	1
3	A	transcode rtmp://192.168.1.172:1935/live/A_src rtmp://192.168.1.172:1935/live/A_400k 400	6	0
4	B	transcode rtmp://192.168.1.172:1935/live/B_src rtmp://192.168.1.172:1935/live/B_2400k 2400	20	0
5	B	transcode rtmp://192.168.1.172:1935/live/B_src rtmp://192.168.1.172:1935/live/B_1600k 1600	16	1
6	B	transcode rtmp://192.168.1.172:1935/live/B_src rtmp://192.168.1.172:1935/live/B_800k 800	14	0
7	C	transcode rtmp://192.168.1.172:1935/live/C_src rtmp://192.168.1.172:1935/live/C_400k 400	6	0

Table 2: Transcoding task examples

the failed task.

3.4 Implementation

We implemented both the cluster manager and transcoders on the Linux operating system. We extensively used open source software in this project, including Apache [3], Nginx [24], node.js + Express [26], mqtt.js [22], paho MQTT client library [21], GStreamer, and Google protobuf. We used Mosquitto [20] as the MQTT message broker.

Media Server: We built the media server with Nginx + Nginx_RTMP_module [25] and Apache. The RTMP protocol is implemented by Nginx, whereas the HTTP interface is provided by Apache. The media server is installed on a server with Ubuntu 14.04 LTS.

Cluster Manager: The cluster manager is written in Javascript, and built on node.js + Express. We use mqtt.js to develop the MQTT client module that subscribes and publishes messages related to the transcoders. The cluster manager is also installed on a server with Ubuntu 14.04 LTS.

Transcoder: The transcoder's two components – transcoder worker and cluster agent are implemented in C/C++. They depend on GStreamer, paho MQTT client library, and Google protobuf. We built the SDK, root disk and Linux kernel for Raspberry Pi with buildroot [4], then we built the two components with the customized SDK.

3.5 Deployment

We deploy VideoCoreCluster in an incremental way. Currently, we leverage a hybrid approach to provide transcoding service for the live video streaming service. The deployment has a small scale VideoCoreCluster with 8 Raspberry Pi Model Bs and a transcoder cluster composed of five powerful servers with Intel Xeon processors. We plan to extend the size of VideoCoreCluster and eventually replace all the servers in the deployment.

4 Evaluations

We evaluate VideoCoreCluster on video quality and transcoding speed with benchmark tests. We also analyze the power consumption of VideoCoreCluster and compare it with a transcoder cluster built with general-purpose processors.

4.1 Video Quality Test

The H.264 decoder module of VideoCore IV can support real-time decoding of H.264 high profile, level 4.0 video with resolutions up to 1920x1080 with very low power consumption. In addition, the decoding result is exactly same as the reference H.264 decoders. So the quality loss of our transcoding system is only from the encoder. Many hardware video encoders have some optimizations to simplify the hardware design while sacrificing video quality, especially for the video encoder modules in SoCs for mobile devices, because of the stringent restriction on power consumption. In order to have a clear idea about the video quality of the VideoCore's H.264 video encoder, we conducted benchmark tests on it and compared its performance with x264.

Both subjective and objective metrics are available for evaluating the video quality. Subjective metrics are desired because they reflect the video quality from users' perspective. However, their measurement procedures are complicated [18, 30]. In contrast, objective metrics are easy to measure; therefore, they are widely used in video encoder developments, even though there are arguments about them. Two metrics, Peak Signal to Noise Ratio (PSNR) and Structural Similarity (SSIM) Index, are

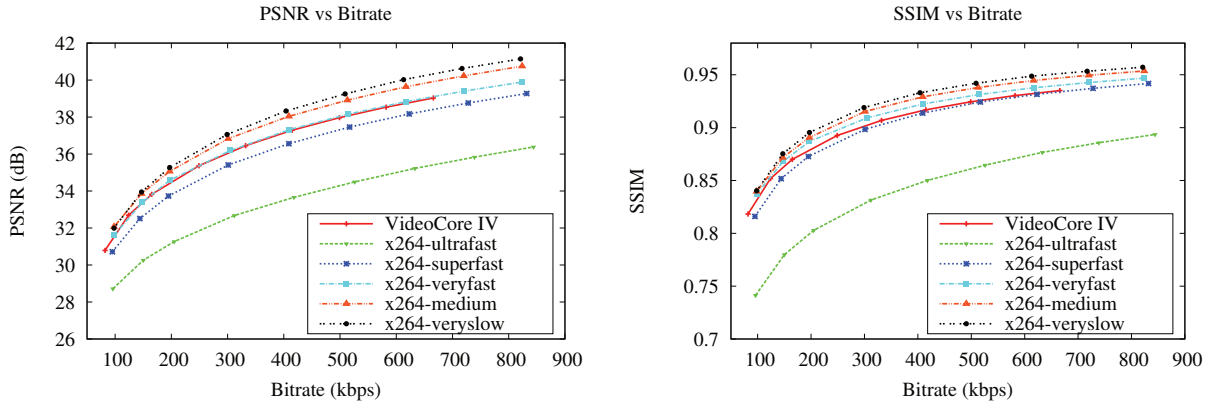


Figure 10: Video quality test result of VideoCore IV and x264 with different presets. We used foreman(352x288, 25fps) YUV sequence to test the encoders, set IDR to 2 seconds, and disabled B-frame support of x264.

widely used to compare video encoders. PSNR is the traditional method, which attempts to measure the visibility of errors introduced by lossy video coding. Huynh-Thu et al. showed that as long as the video content and the codec type are not changed, PSNR is a valid quality measure [50]. SSIM is a complementary framework for quality assessment based on the degradation of structural information [63]. We evaluated the hardware video encoder's performance with both the PSNR and SSIM.

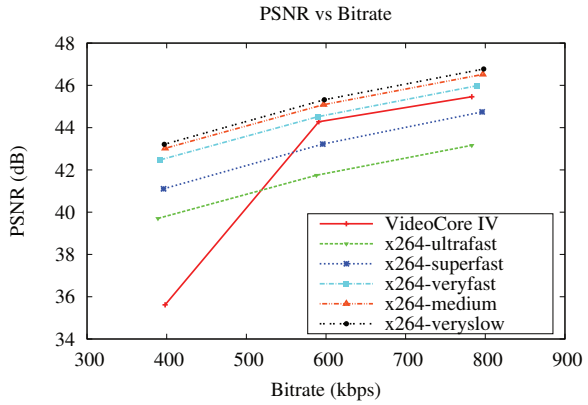
The x264 has broad parameters to tune, which are correlated and it is hard to achieve the optimal configuration. Rather than trying different parameter settings, we used the presets provided by x264 developers to optimize the parameters related to video encoding speed and video quality. The x264 has ten presets: ultrafast, superfast, veryfast, faster, fast, medium, slow, slower, veryslow, and placebo (the default preset is medium). They are in descending order of speed and ascending order of video quality. As the video quality increases, encoding speed decreases exponentially.

We used two sets of video sequences for evaluating the encoders. The first one is the YUV sequences commonly used in video coding research [39], so the results can be easily reproduced and compared with other researcher's results. The second one is captured from our deployment that reflects the encoder's performance in practical deployment. We natively compiled the x264-snapshot-20150917-2245 on a desktop with Ubuntu 14.04 LTS. We also cross-compiled the libraries, drivers and firmware for Raspberry Pi from [33, 35] on the same machine. One thing we notice is that the H.264 encoder of VideoCore IV does not support B-frames. The reason is that the target applications of the SoC are real-time communication applications, e.g., video chatting and conference, where low latency is a strict requirement. B-frame leads to high encoding/decoding latency. Thus, the H.264 encoder of VideoCore IV does not support it. For a fair comparison, we test x264 with and without B-frame support to check the impact of B-frame support on video quality vs. bitrate.

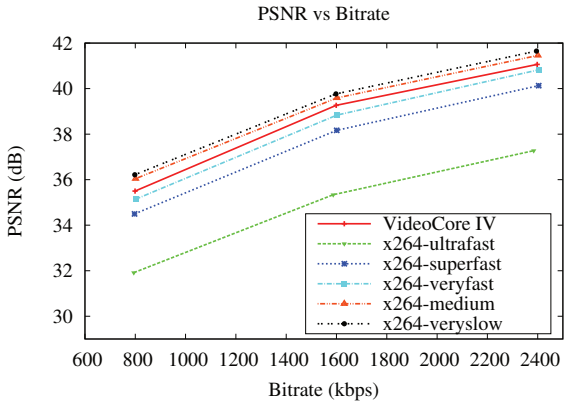
For all the video sequences we tested in the first set, we obtained similar results, though the exact numbers vary. We also found that the VideoCore's video encoder generated very low-quality video when the target bitrate was very low. That could be a bug in the video encoder's implementation. For brevity, we only show the results for foreman_cif here. We omit the results of x264 with B-frame because B-frame does not have a significant impact on the quality in our encoding settings. From figure 10, we can see VideoCore IV has similar or better performance regarding video quality comparing to the x264 with preset superfast. Since the purpose of the second test set is to evaluate the video encoder's performance in practical deployment, we only evaluated the encoder's performance with the typical bitrates. Figures 11a and 11c indicate that the VideoCore has poor performance on low bitrate settings. However, we believe it is not a big concern for deployment. When the players have to use that low bitrate version of the video streams, the player's network performance must be very low. We do not expect that will be a common condition. As shown in Figures 11b and 11d, VideoCore's video quality is good for high bitrate settings. Its quality is close to x264 with preset medium.

4.2 Transcoding Speed Test

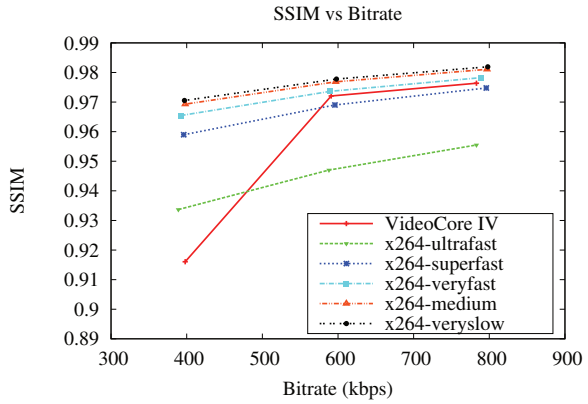
We measured the transcoder's performance under stress. We transcoded the video streams captured from an SD channel and an HD channel offline and recorded the time for transcoding. There are overheads on demuxing and muxing in the process, but because of the high complex-



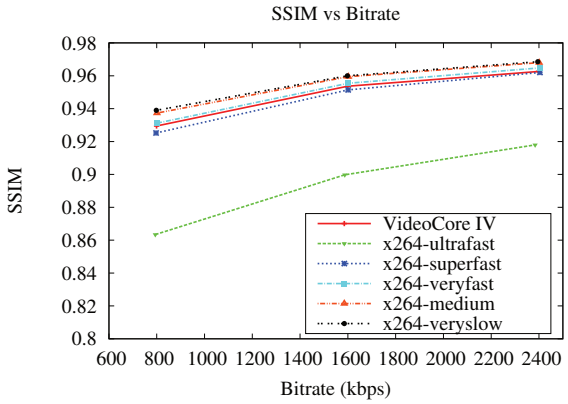
(a) PSNR values of an SD channel (720x480, 30fps)



(b) PSNR values of an HD channel (1280x720, 30fps)



(c) SSIM values of an SD channel (720x480, 30fps)



(d) SSIM values of an HD channel (1280x720, 30fps)

Figure 11: Video quality test result of VideoCore IV and x264 with different presets. We maintained the same configurations (IDR is 2 seconds, B-frame support of x264 is disabled).

ity of transcoding (decoding + encoding), the bottleneck of the process is on video transcoding. The detail specifications of the test videos are (1) SD channel: 720x480, 30fps, H.264 high profile, level 4.0, 1.2 Mbps. (2) HD channel: 1280x720, 30fps, H.264 high profile, level 4.0, 4Mbps. For some SD channels with lower resolutions in our deployment, the transcoding speed is higher than the results shown here. We transcoded the SD and HD videos to 800kbps and 2.4Mbps respectively and kept other parameters the same.

The hardware video encoder in VideoCore IV can support encoding 1920x1080, 30fps, H.264 high profile video in real-time. But when we run the decoder and encoder at the same time, the performance is not sufficient to support such high-resolution transcoding in real-time because the video decoder and encoder share some hardware resources. Even with the optimization described in section 3, the transcoder can only support transcoding video with resolution up to 1280x720 in real-time.

We also measured software transcoder's speed for comparison. The software video transcoder we used is FFmpeg, which has built-in H.264 video decoder. We linked it with libx264 to provide H.264 video encoding. The desktop we used to run FFmpeg has Intel Core i5-4570 CPU @ 3.20GHz and 16GB RAM. We built FFmpeg and libx264 with all the CPU capabilities (MMX2, SSE2Fast, SSSE3, SSE4.2, AVX, etc.) to accelerate the video transcoding. We tested superfast (similar quality as the video encoder of VideoCore IV) and medium (default) presets of x264. Figure 12 shows that when the output video qualities are similar, software transcoder executing on powerful Intel i5 CPU runs about 5.5x and 4x faster than the video transcoder running on Raspberry Pi for SD and HD channel respectively. For our transcoding system, which transcodes video in real-time, that means we can run 5.5x (SD) or 4x (HD) more transcoding tasks on a desktop than a Raspberry Pi. However, a Raspberry Pi is much cheaper and consumes much less power than a desktop with an Intel i5 CPU.

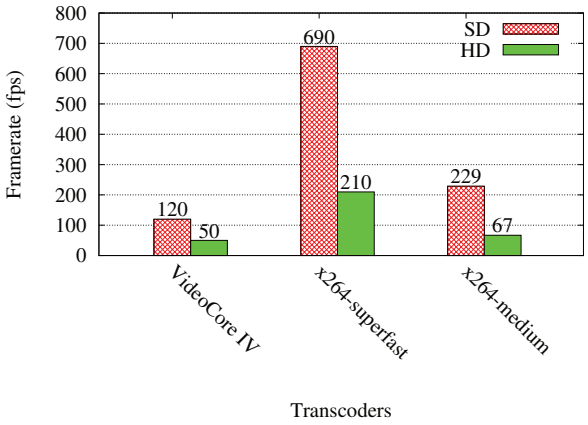


Figure 12: Transcoding speed of VideoCore IV and x264.

4.3 Power Consumption Analysis

We can see the superiority of Raspberry Pi regarding power efficiency from the transcoding speed test. An Intel Core i5-4570 processor has an average power consumption of 84W [14]. If we include the power consumption of other components, e.g., RAM and Hard Disk, the power consumption of a server would be higher than 100W. Raspberry Pi Model B in a configuration without any peripherals except Ethernet has typical power consumption about 2.1W [45]. The desktop consumes more than 40 times power than the Raspberry Pi Model B while it can do about 5.5x SD video transcoding or 4x HD video transcoding. We can see the VideoCoreCluster is more energy efficient than a transcoder cluster built with general-purpose processors to provide the same transcoding capacity. We omit the power consumption analysis on the network switches in our system deployment because we can use the same switches for the different video transcoders.

5 Related Work

Video Transcoding: Video transcoding is critical for video streaming service deployment. Different approaches and architectures have been proposed to implement it for various use cases. Vetro et al. discussed the transcoding of block-based video coding schemes that use hybrid discrete cosine transform (DCT) and motion compensation (MC) [62]. Xin et al. discussed several techniques for reducing the complexity and improving video quality by exploiting the information extracted from the input video bit stream [65]. Unlike their research, we believe that the cascaded decoder and encoder approach is much more straightforward and flexible. As the hardware video encoder improving quality and efficiency and reducing the cost, a cascaded pixel-domain approach is more suitable for practical deployments. For a particular scenario, Youn et al. showed that for point-to-multipoint transcoding, a cascaded video transcoder is more efficient since some parts of the transcoder can be shared [67].

Cloud Transcoding: Li et al. introduced a system using cloud transcoding to optimize video streaming service for mobile devices [54]. Video transcoding on a cloud platform is a good solution to transcode a large volume of video data because of its high throughput. For instance, Amazon, Microsoft, and Telestream Cloud provide cloud transcoding service for users [19, 1, 12]. Netflix also deployed their video transcoding platform on Amazon's cloud [23].

Specialized Hardware for Video Applications: Efficiency issue of general-purpose processors on multimedia applications, including video decoding and encoding, has attracted a lot of research efforts. Various approaches have been studied to improve the hardware efficiency, including specialized instructions [58, 44], specialized architectures [52, 59, 64], GPU offloading [55, 43], application-specific integrated circuit(ASIC) [56], and FPGA-based accelerators [53].

Adaptive Bitrate Video Streaming: Adaptive bitrate video streaming is a widely used technique by video streaming service providers to provide high-quality video streaming services. Designing a robust and reliable algorithm to switch bitrate is challenging. Many researchers have proposed adaptation algorithms to achieve a better video quality in dynamic network environments [51, 49, 66]. These works focus on the client side implementation, whereas our paper concentrates on the server side.

Computer Cluster: Computer cluster is a well-known scheme of distributed system used to provide high throughput computing. For example, Condor is a distributed system for scientific applications [61]. Our system is unique in the sense that the target application is real-time computing, and the computing nodes are specialized, low cost and highly efficient hardware. FAWN is also a cluster built with low-power embedded CPUs. However, it is a system only for the data-intensive computing [41]. Our system is both the data-intensive and computation-intensive.

6 Conclusion and Future Work

High-quality video transcoding is critical to ensure high-quality video streaming service. We implemented VideoCoreCluster, a low-cost, highly efficient video transcoder system for live video streaming service. We built the system with commodity available, low-cost single board computers embedding high performance and low power

video encoder hardware module. We implemented the cluster based on a network protocol for IoT for the control path and RTMP for the data path. This separation design can get low latency in video transcoding and data delivery. Our system has much higher energy efficiency than the transcode cluster built with general-purpose processors, and it does not sacrifice quality, reliability, or scalability. We can use VideoCoreCluster on other live video streaming services, and we can further improve the system on capability and energy efficiency by upgrading the transcoders.

7 Acknowledgements

We thank Derek Meyer for his help in the system deployment. We are grateful to our shepherd, Anthony Joseph, and the anonymous reviewers whose comments helped bring the paper to its final form. All authors are supported in part by the US National Science Foundation through awards CNS-1555426, CNS-1525586, CNS-1405667, CNS-1345293, and CNS-1343363.

References

[1] Amazon elastic transcoder. https://aws.amazon.com/elastictranscoder/.

[2] Ambarellas broadcast infrastructure solutions. http://www.ambarella.com/products/broadcast-infrastructure-solutions#S3.

[3] Apache http server project. https://httpd.apache.org/.

[4] Buildroot - making embedded linux easy. https://buildroot.org/.

[5] Cisco visual networking index: Forecast and methodology, 2014-2019 white paper. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.

[6] Eighth mpeg-4 avc/h.264 video codecs comparison - standard version. http://www.compression.ru/video/codec_comparison/h264_2012/.

[7] Enabling the full 4k mobile experience: System leadership. https://www.qualcomm.com/documents/enabling-full-4k-mobile-experience-system-leadership.

[8] gst-omx. https://github.com/pliu6/gst-omx.

[9] Gstreamer: open source multimedia framework. http://gstreamer.freedesktop.org/.

[10] H.264 4k video encoder. http://www.xilinx.com/products/intellectual-property/1-4iso3h.html.

[11] H.264/avc reference software. http://iphome.hhi.de/suehring/tml/download/.

[12] High quality video transcoding in the cloud. https://cloud.telestream.net/.

[13] Http live streaming. https://developer.apple.com/streaming/.

[14] Intel core i5-4570 processor. http://ark.intel.com/products/75043/Intel-Core-i5-4570-Processor-6M-Cache-up-to-3_60-GHz.

[15] Intel integrated performance primitives. https://software.intel.com/en-us/intel-ipp.

[16] Introducing the video coding engine (vce). http://developer.amd.com/community/blog/2014/02/19/introducing-video-coding-engine-vce/.

[17] Mainconcept. http://www.mainconcept.com/.

[18] Methodology for the subjective assessment of the quality of television pictures. https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.500-13-201201-I!!PDF-E.pdf.

[19] Microsoft azure media services. https://azure.microsoft.com/en-us/services/media-services/encoding/.

[20] Mosquitto - an open source mqtt v3.1/v3.1.1 broker. http://mosquitto.org/.

[21] Mqtt c++ client for posix and windows. https://eclipse.org/paho/clients/cpp/.

[22] Mqtt.js. https://github.com/mqttjs.

[23] Netflix's encoding transformation. http://www.slideshare.net/AmazonWebServices/med202-netflixtranscodingtransformation.

[24] Nginx. https://www.nginx.com/.

[25] nginx-rtmp-module. https://github.com/pliu6/nginx-rtmp-module.

[26] Node.js. https://nodejs.org/en/.

[27] Nvidia video codec sdk. https://developer.nvidia.com/nvidia-video-codec-sdk.

[28] Openh264. http://www.openh264.org/.

[29] Openmax integration layer application programming interface specification. https://www.khronos.org/registry/omxil/specs/OpenMAX_IL_1_1_2_Specification.pdf.

[30] P.910 : Subjective video quality assessment methods for multimedia applications. https://www.itu.int/rec/T-REC-P.910/en.

[31] Programming audiovideo on the raspberry pi gpu. https://jan.newmarch.name/RPi/.

[32] Protocol buffers. https://developers.google.com/protocol-buffers/?hl=en.

[33] Raspberry pi firmware. https://github.com/raspberrypi/firmware.

[34] Real-time messaging protocol (rtmp) specification. http://www.adobe.com/devnet/rtmp.html.

[35] Source code for arm side libraries for interfacing to raspberry pi gpu. https://github.com/raspberrypi/userland.

[36] State of the art of video on smartphone. http://ngcodec.com/news/2014/3/13/state-of-the-art-of-video-on-smartphone.

[37] To stream everywhere, Netflix encodes each movie 120 times. https://gigaom.com/2012/12/18/netflix-encoding/.

[38] x264. http://www.videolan.org/developers/x264.html.

[39] Yuv video sequences. http://trace.eas.asu.edu/yuv/.

[40] AHMAD, I., WEI, X., SUN, Y., AND ZHANG, Y.-Q. Video transcoding: an overview of various techniques and research issues. *Multimedia, IEEE Transactions on 7*, 5 (2005), 793–804.

[41] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 1–14.

[42] BANKS, A., AND GUPTA, R. Mqtt version 3.1. 1. *OASIS Standard* (2014).

[43] CHEN, W.-N., AND HANG, H.-M. H. 264/avc motion estimation implmentation on compute unified device architecture (cuda). In *Multimedia and Expo, 2008 IEEE International Conference on* (2008), IEEE, pp. 697–700.

[44] DAIGNEAULT, M.-A., LANGLOIS, J. P., AND DAVID, J. P. Application specific instruction set processor specialized for block motion estimation. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on* (2008), IEEE, pp. 266–271.

[45] DICOLA, T. Embedded linux board comparison. `https://learn.adafruit.com/downloads/pdf/embedded-linux-board-comparison.pdf`.

[46] DRAFT, I. recommendation and final draft international standard of joint video specification (itu-t rec. h. 264— iso/iec 14496-10 avc). *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVTG050 33* (2003).

[47] GEBERT, S., PRIES, R., SCHLOSSER, D., AND HECK, K. Internet Access Traffic Measurement and Analysis. In *Proc. of ACM TMA* (2012).

[48] HAMEED, R., QADEER, W., WACHS, M., AZIZI, O., SOLOMATNIKOV, A., LEE, B. C., RICHARDSON, S., KOZYRAKIS, C., AND HOROWITZ, M. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 37–47.

[49] HUANG, T.-Y., JOHARI, R., MCKEOWN, N., TRUNNELL, M., AND WATSON, M. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 187–198.

[50] HUYNH-THU, Q., AND GHANBARI, M. Scope of validity of psnr in image/video quality assessment. *Electronics letters 44*, 13 (2008), 800–801.

[51] JIANG, J., SEKAR, V., AND ZHANG, H. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 97–108.

[52] KIM, S. D., AND SUNWOO, M. H. Asip approach for implementation of h.264/avc. *Journal of Signal Processing Systems 50*, 1 (2008), 53–67.

[53] LEHTORANTA, O., SALMINEN, E., KULMALA, A., HÄNNIKÄINEN, M., AND HÄMÄLÄINEN, T. D. A parallel mpeg-4 encoder for fpga based multiprocessor soc. In *Field Programmable Logic and Applications, 2005. International Conference on* (2005), IEEE, pp. 380–385.

[54] LI, Z., HUANG, Y., LIU, G., WANG, F., ZHANG, Z.-L., AND DAI, Y. Cloud transcoder: Bridging the format and resolution gap between internet videos and mobile devices. In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video* (2012), ACM, pp. 33–38.

[55] LIN, Y.-C., LI, P.-L., CHANG, C.-H., WU, C.-L., TSAO, Y.-M., AND CHIEN, S.-Y. Multi-pass algorithm of motion estimation in video encoding for generic gpu. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on* (2006), IEEE, pp. 4–pp.

[56] LIN, Y.-K., LI, D.-W., LIN, C.-C., KUO, T.-Y., WU, S.-J., TAI, W.-C., CHANG, W.-C., AND CHANG, T.-S. A 242mw, 10mm 2 1080p h. 264/avc high profile encoder chip. In *Proceedings of the 45th annual Design Automation Conference* (2008), ACM, pp. 78–83.

[57] MERRITT, L., AND VANAM, R. Improved rate control and motion estimation for h. 264 encoder. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on* (2007), vol. 5, IEEE, pp. V–309.

[58] PELEG, A., AND WEISER, U. Mmx technology extension to the intel architecture. *Micro, IEEE 16*, 4 (1996), 42–50.

[59] SEO, S., WOH, M., MAHLKE, S., MUDGE, T., VIJAY, S., AND CHAKRABARTI, C. Customizing wide-simd architectures for h.264. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on* (2009), IEEE, pp. 172–179.

[60] STOCKHAMMER, T. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems* (2011), ACM, pp. 133–144.

[61] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: The condor experience. *Concurrency-Practice and Experience 17*, 2-4 (2005), 323–356.

[62] VETRO, A., CHRISTOPOULOS, C., AND SUN, H. Video transcoding architectures and techniques: an overview. *Signal Processing Magazine, IEEE 20*, 2 (2003), 18–29.

[63] WANG, Z., BOVIK, A. C., SHEIKH, H. R., AND SIMONCELLI, E. P. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on 13*, 4 (2004), 600–612.

[64] WOH, M., SEO, S., MAHLKE, S., MUDGE, T., CHAKRABARTI, C., AND FLAUTNER, K. Anysp: anytime anywhere anyway signal processing. In *ACM SIGARCH Computer Architecture News* (2009), vol. 37, ACM, pp. 128–139.

[65] XIN, J., LIN, C.-W., AND SUN, M.-T. Digital video transcoding. *Proceedings of the IEEE 93*, 1 (2005), 84–97.

[66] YIN, X., JINDAL, A., SEKAR, V., AND SINOPOLI, B. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 325–338.

[67] YOUN, J., XIN, J., SUN, M.-T., AND ZHANG, Y.-Q. Video transcoding for multiple clients. In *Visual Communications and Image Processing 2000* (2000), International Society for Optics and Photonics, pp. 76–85.

Notes

[1] During the course of this work, J. Yoon was a graduate student at the University of Wisconsin-Madison, and L. Johnson was a staff at the University of Wisconsin-Madison.

MEANTIME: Achieving Both Minimal Energy and Timeliness with Approximate Computing

Anne Farrell *Henry Hoffmann*
Department of Computer Science, University of Chicago
{amfarrell, hankhoffmann}@cs.uchicago.edu

Abstract

Energy efficiency and timeliness (*i.e.,* predictable job latency) are two essential – yet opposing – concerns for embedded systems. Hard timing guarantees require conservative resource allocation while energy minimization requires aggressively releasing resources and occasionally violating timing constraints. Recent work on approximate computing, however, opens up a new dimension of optimization: application accuracy. In this paper, we use approximate computing to achieve both hard timing guarantees and energy efficiency. Specifically, we propose MEANTIME: a runtime system that delivers hard latency guarantees and energy-minimal resource usage through small accuracy reductions. We test MEANTIME on a real Linux/ARM system with six applications. Overall, we find that MEANTIME never violates real-time deadlines and sacrifices a small amount (typically less than 2%) of accuracy while reducing energy to 54% of a conservative, full accuracy approach.

1 Introduction

Embedded systems require both predictable timing and energy-efficiency. When predictability takes the form of hard real-time constraints, these two goals are conflicting [16]. The conflict arises because hard timing guarantees require reserving resources sufficient for worst case latency. In contrast, energy efficiency requires allocating resources that just meet the needs of the current job. Even if worst case resource allocation is coupled with aggressive energy reduction (*e.g.,* in the form of voltage scaling [20] or sleep states [31]), this strategy is less energy-efficient than allocating for the current case, a fact which has been demonstrated both analytically [1, 8, 35] and empirically [17, 32, 38, 45].

Recent research on *approximate computing* examines applications that can trade *accuracy* for reduced energy consumption [4, 6, 18, 21, 29, 55]. These approximate computations open a third dimension for optimization, making it possible to simultaneously trade timing, energy, and accuracy [26]. Many embedded computations are well-suited for approximation as they involve large amounts of signal, image, media, and data processing, where it is easy to quantify application accuracy (*e.g.,* as signal-to-noise-ratio) and carefully trade it for other benefits. This process is often done statically; *e.g.,* by using fixed point arithmetic instead of floating point. While approximate computing frameworks produce a large trade-off space [4, 29, 51], they provide (at best) soft timing guarantees, and are thus unsuitable for hard real-time constraints. *This paper addresses the challenge of meeting both hard timing constraints and low energy through careful application of approximate computing.*

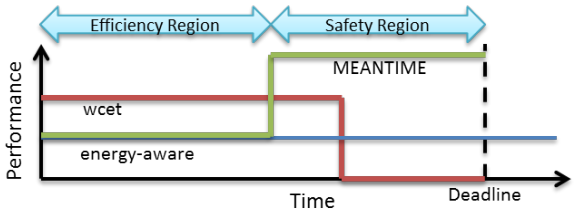


Figure 1: Conceptual model of MEANTIME compared to worst-case and energy-aware resource allocation.

Specifically, we develop MEANTIME[1], which couples a state-of-the-art resource allocator [42] – that optimizes for the current job, sometimes missing deadlines – with a novel *governor* which monitors timing and reconfigures the application to avoid any timing violations. The resource allocator uses control theory to allocate for the current case, an approach which, by itself, will violate timing constraints. Therefore, the governor determines how much to manipulate application accuracy to ensure that the timing constraints are never violated despite the dynamics of the underlying controller and any application-level fluctuations.

Figure 1 illustrates the intuition behind MEANTIME

[1]The name stands for Minimal Energy ANd TIMEliness.

Table 1: Embedded Platform Resources.

Platform	Processor	Cores	Core Types	Speeds (GHz)	Configurations
ODROID-XU3	Samsung Exynos5 Octa	8	2 (A15 & A7)	.2–2.0 (A15) .2-1.4 (A7)	128

and compares it to other resource allocation approaches. The figure shows time, with a deadline, on the x-axis and performance on the y-axis. Allocating for worst case execution time (wcet) requires using all resources in the system and then idling (no performance) until the deadline. Energy-aware allocation estimates the resource needs of the current job, but it may miss the deadline due to job timing variance.

In contrast, MEANTIME reasons about wcet for both the application's nominal behavior and its acceptable approximate variants. Given this information, MEANTIME allocates for the current case, while computing two temporal regions within the deadline: an *efficiency region* and a *safety region* (shown in Figure 1). The efficiency region represents the time to run in the application's full-accuracy configuration using the resources specified by the energy-aware allocator. The safety region represents the time for which the application must switch to an approximate configuration (still using the assigned resources). Note that switching to an approximate configuration does not require computation to be restarted; rather there are times while an application is running that it can be switched to an approximate configuration before the deadline. Approximation allows MEANTIME to potentially have even higher performance in the safety region than allocating for worst case execution time, as shown in Figure 1. MEANTIME's key idea is using timing analysis and approximate computation to determine the efficiency and safety regions, ensuring both energy efficiency and timeliness.

We implement MEANTIME on a Linux/ARM system. We test its timing guarantees, energy, and accuracy for six different applications, including media, image, signal processing, and financial analysis. These applications were not originally designed to provide any timing predictability, still MEANTIME achieves:

- **Efficacy with a range of behaviors**: The benchmarks include both high and low variance job latencies (see Section 4.3).
- **Predictable timing**: MEANTIME never misses a deadline for any benchmark (see Section 5.1).
- **Energy Savings**: MEANTIME requires only 46% of the energy of allocating for worst case and aggressively sleeping the system (see Section 5.2). These energy savings are comparable to a state-of-the-art energy-aware approach that provides only soft timing guarantees but near-optimal energy savings.
- **High Accuracy**: MEANTIME achieves accuracies that are typically very close to the nominal behavior (*i.e.,* almost no accuracy loss, see Section 5.3).
- **Adaptability**: MEANTIME automatically reacts to changing user goals (see Section 5.4) and fluctuations in application workload (see Section 5.5).

MEANTIME is not designed for all embedded applications, but for those that 1) have viable performance/accuracy trade-offs, 2) must satisfy hard real-time constraints and minimize energy consumption despite large fluctuations in application workload, and 3) have progress indicators and models of completion. We believe many embedded applications meet these criteria (as evidenced by our use of existing benchmarks).

MEANTIME makes the following contributions:

- Design of a runtime system that provides both hard real-time guarantees and energy efficiency by sacrificing computation accuracy.
- Experimental evaluation of the runtime on a real system with six different benchmarks showing mean energy reductions of over 2× compared to allocating for worst case.

2 Motivational Example

We motivate MEANTIME's combination of hard timing guarantees and energy efficiency through a radar processing example similar to what might be found on an unmanned autonomous vehicle (UAV). The radar must process frames with a strict latency or the system cannot respond to external events. The UAV, however, is energy limited. Radar processing is amenable to approximate computation – it can trade reduced signal-to-noise ratio for performance.

The radar processing consists of four *blocks* of processing. The first block (LPF) performs a low-pass filter to eliminate high-frequency noise. The second block (BF) does beam-forming, steering the radar to "look" in a particular direction. The third block (PC) performs pulse compression, which concentrates energy. The final block is constant false alarm rate (CFAR) detection, which identifies targets. Several parameters control the tradeoff between signal-to-noise ratio (SNR) and performance (see Section 4). The radar processes frames in batches of 8. Each batch is one job.

Our experimental platform is an 8-core Linux/ARM big.LITTLE system, which has four high power, high performance "big" cores, and four low power, low performance "LITTLE" cores. The radar application is parallelized, so that each processing block can be executed

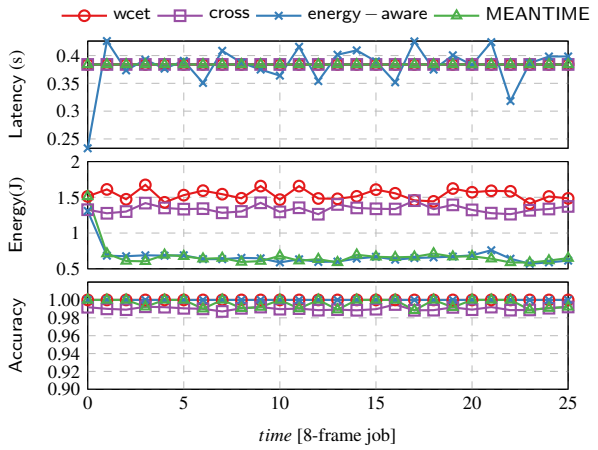


Figure 2: Comparison of techniques for the radar.

across multiple threads. Different resource configurations achieve different power/performance tradeoffs; Table 1 lists available configurations.

We run the radar application on the ARM with all resources allocated and empirically determine the worst case latency for a single radar frame (this is technically worst *observed* latency). Additionally, we report the minimum, mean, and standard deviation over mean latency in Table 2.

Table 2: Radar timing.

Latency	Measurement (s)
Mean	0.032
Minimum	0.031
Maximum	0.048
STDEV/Mean	0.025

The timing data demonstrates the classic conflict between timeliness and energy efficiency. For timeliness, we can allocate for worst-case execution time. The table, however, shows that the average latency is very close to the minimum latency. So most jobs are not close to worst case. If a job finishes early (*i.e.,* it is not a worst case), then we can simply sleep (or idle) the processor until the next job is available [17, 38, 45]. To allocate for energy efficiency and meet average case timing, we could use a state of the art energy-aware approach (*e.g.,* [42]).

The key idea behind MEANTIME is using energy-aware techniques to allocate for the average case, but avoid timing violations by quickly switching the application to an approximate configuration that is guaranteed to meet timing even for a worst case frame. Section 3 discusses in detail how MEANTIME calculates the times to spend in different configurations.

We demonstrate the benefits of MEANTIME for the radar application by comparing its latency, energy, and accuracy to three other approaches: allocating resources for worst case (wcet), allocating both resources and algorithm parameters for worst case based on existing *cross-layer* approaches (cross) [21, 58], and allocating for average case using an *energy-aware* resource allocator based on control theory [33, 42]. Figure 2 shows the results. Each chart shows time (measured in 8-frame jobs) on the x-axis. The top chart shows latency compared to the target latency (0.384 seconds per 8-frame job, the worst latency measured on our system). The middle chart shows the energy per job in Joules. The bottom chart shows the SNR normalized to the default configuration; *i.e.,* unity represents no accuracy loss.

The results demonstrate that MEANTIME achieves both timeliness and energy efficiency. Indeed, MEANTIME achieves the same timing as allocating for worst case, the same energy consumption as the energy-aware system, and the same accuracy as the cross-layer approach. All approaches except for the energy-aware approach have flat latency curves because they always meet the deadline (possibly finishing early and idling until the next job starts) – these techniques are hard to distinguish in the latency chart because all three are on top of each other.. Specifically, the energy-aware approach consumes 20.2 Joules, allocating for wcet consumes 46.1 Joules, the cross-layer approach consumes 36.7 Joules, and MEANTIME consumes 20.3 Joules. These results come at the cost of a small amount of accuracy; in this case the SNR falls to 99.7% of the original application.

3 The MEANTIME Framework

This section provides the technical details on MEANTIME, illustrated in Figure 3. As shown in the figure, MEANTIME couples a *governor* with an *energy-aware* controller. The controller allocates system resources (*e.g.,* cores, clockspeed, memory bandwidth) and the governor ensures that the application's timing requirements are met. The controller receives a latency target from the governor and computes the minimal energy set of resources needed to meet that target. The governor receives, from the user, timing requirements and application's timing analysis in both its default and approximate configurations. The user has complete control over which set of approximations MEANTIME considers. The governor translates these timing requirements and analysis into a latency target, which is passed to the controller. The governor calculates the *efficiency* and *safety* regions. The efficiency region is the time to spend in the application's default configuration using the resource configuration suggested by the controller. The safety region is the time to spend in an approximate configuration, ensuring the job meets its deadline.

This section describes MEANTIME's controller and governor. The controller itself is not a contribution of MEANTIME; we make use of existing research on energy-aware control for soft real-time requirements. MEANTIME's contributions are 1) the governor which ensures timeliness, and 2) the integration of the gover-

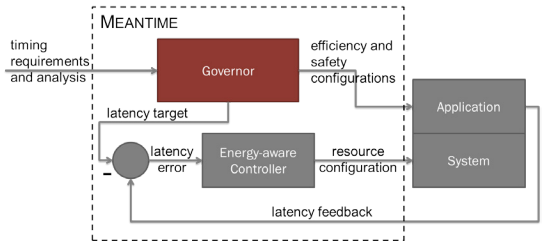


Figure 3: Overview of the MEANTIME approach.

nor with existing control techniques. The remainder of this section provides an overview of energy-aware control schemes, then describes the governor.

3.1 Approximate Computing

Many approximate computing frameworks exist (see Section 6) that expose application-level tradeoffs between accuracy and resource usage (almost always quantified as execution time). Execution time is straightforward. Accuracy, however, is defined in an application-specific manner. In a radar, accuracy is signal-to-noise ratio. In a search engine, it is the precision and recall of results returned. We discuss the accuracy metrics used for our test applications in Section 4. The essential thing for MEANTIME is that the framework exposes the performance and accuracy tradeoffs for a particular application. Performance gains can be quantified in a standard way. Accuracy need only be a total order; MEANTIME works even if the exact accuracy of a configuration cannot be specified, as long as it can be ranked relative to other configurations. MEANTIME requires this ordering so that it can choose the highest accuracy configuration that meets a constraint.

3.2 Control for Energy Efficiency

Control theory provides formal guarantees about how a controlled system responds to dynamic events [24, 59]. Control theoretic techniques have been used to meet soft real-time deadlines while minimizing resource use. Examples can be found controlling latency in multi-tier webservers [11, 30] and in media processing [56, 58].

To minimize energy for a latency constraint, the controller is given a target latency. It then measures the current latency, computes the error between the target and current, and determines the most efficient resource allocation that will eliminate the error. Different control implementations have different tradeoffs between how fast they react and how stable they are in the face of noise. The important thing for MEANTIME is that control approaches have repeatedly proven capable of near optimal resource allocation [11, 25, 30, 33, 58].

The drawback is that these techniques, by themselves, cannot provide both hard real-time latency and energy efficiency. One might attempt to use control theoretic techniques to meet hard real-time constraints by reducing the target latency, but this does not guarantee hard real-time and may not provide optimal energy savings. MEANTIME, therefore, relies on a control system to allocate resources for the current case and minimize energy. Several existing control systems might be appropriate for this task, but MEANTIME builds on POET, a portable, open-source energy-aware control implementation [33]. The concept is general, however, and could be applied to many different control systems.

Within MEANTIME, the controller is responsible for reacting to application phases; *e.g.,* reducing resource usage if the workload gets easier and increasing resources if it gets harder. Control systems are well-suited to this task, as their whole purpose is managing system dynamics. Of course, control systems react to changes by detecting errors between a target latency and an achieved latency. Such latency errors represent missed deadlines and are not acceptable for hard real-time. Therefore, MEANTIME's governor ensures that the user's target latency requirement is not violated.

3.3 Governor for Timeliness

The governor's primary responsibility is deriving the efficiency and safety regions. To perform this task, the governor requires the following inputs:

1. Worst case timing for the application in its full accuracy configuration.
2. Worst case timing (expressed as speedup) for reduced accuracy configurations.
3. Worst case timing analysis for switching application or system configurations.

While timing information may vary, we only require information for the worst case, which is conservative and does not vary. This section shows how the governor derives the efficiency and safety regions from these inputs.

3.3.1 Notation

Table 3 summarizes the notation used in this section. We assume an application comprised of many jobs, where each has a *workload* representing its processing requirements. We assume we know the worst case workload W and the deadline for completing the work t. We label the worst case latency and computation rate as t_{wc} and r_{wc} and best case values are t_{bc} and r_{bc}. Internally, MEANTIME records the relationship between the best and worst case as $\Delta = t_{bc}/t_{wc}$. Note that $0 < \Delta \leq 1$.

Both the machine and application are configurable. The machine's resources can be allocated and the application's accuracy can be changed. Application speedups

Table 3: Notation.

	Variable	Meaning
Input	W	job workload in worst case
	t	deadline for completing work
	t_{wc}	worst case latency
	t_{bc}	best case latency
	r_{wc}	worst case computation rate with full accuracy
	r_{bc}	best case computation rate with full accuracy
	s_0	minimum speedup from approximation
	t_{switch}	worst case time to switch app. & sys. config.
Internal	t_s	time to spend in safety region
	t_e	time to spend in efficiency region
	r_s	computation rate in safety region
	r_e	computation rate in efficiency region
	Δ	ratio of best to worst case, t_{bc}/t_{wc}

are measured relative to the full accuracy application running with all machine resources. We assume we know s_0, the maximum worst case speedup available from changing application accuracy; *i.e.,* the minimum speedup that will be observed from approximation.

3.3.2 Goal

Given the above assumptions and notation, the governor computes the safety and efficiency regions such that the workload of all jobs is completed by their deadlines. We write these requirements as three constraints:

$$t_s \cdot r_s + t_e \cdot r_e \geq W \quad (1)$$

$$t_s + t_e \leq t \quad (2)$$

$$t_s,\ t_e \geq 0 \quad (3)$$

Eqn. 1 states that total capacity for work should be greater or equal to the worst case workload[2]. This constraint is conservative, but if it holds then we know the worst case work will be accomplished. The second constraint ensures that the total time is less than or equal to the deadline time. The third constraint ensures that the times are non-negative, which means the deadline can be met in practice. The governor determines the values of t_e and t_s such that the constraints will be respected, ensuring hard real-time guarantees.

3.3.3 Deriving Efficiency and Safety Regions

MEANTIME considers the most difficult situation, which occurs when the application transitions from a best case latency job to a worst case job. The controller will detect the best case and allocate a commensurate amount of resources. Thus, the controller allocated resources for best case. The combination of worst case workload and resources allocated for best case will severely degrade measured performance. MEANTIME calculates this degraded performance as Δr_{wc}. Therefore, the computation rate in the efficiency region can be as low as $r_e = \Delta r_{wc}$. Furthermore, we can rewrite r_s in terms of known quantities by recognizing that the worst case speed in the safety region is $r_s = \Delta r_{wc} \cdot s_0$. To ensure the constraints are satisfied even in this situation, we substitute into Eqns. 1 and 2 to obtain:

$$t_s \cdot \Delta r_{wc} \cdot s_0 + t_e \cdot \Delta r_{wc} = W \quad (4)$$

$$t_s + t_e + t_{switch} = t \quad (5)$$

We now have a system of two equations with two unknowns: t_s and t_e, the time to spend in the safety and efficiency regions, respectively. All other quantities are known based on timing analysis as stated in the above assumptions. We therefore rewrite Eqn. 5 as $t_s = t - t_e - t_{switch}$ and substitute into Eqn. 4 to obtain:

$$W = (t - t_e - t_{switch}) \cdot \Delta r_{wc} \cdot s_0 + t_e \cdot \Delta r_{wc} \quad (6)$$

$$t_{wc} = \frac{W}{\Delta r_{wc}} = (t - t_e - t_{switch}) \cdot s_0 + t_e \quad (7)$$

$$t_e = \frac{t_{wc} - s_0 \cdot (t - t_{switch})}{1 - s_0} \quad (8)$$

Thus, Eqn. 8 gives the efficiency region; *i.e.,* the largest amount of time we can spend using the resources specified by the controller. We then transition to the variable accuracy configuration (keeping resource usage the same) for $t_s = t - t_e - t_{switch}$ time. A negative value of either t_s or t_e indicates that the application is not schedulable.

An additional quick check for schedulability can be done by checking whether $s_0 \geq \frac{1}{\Delta}$. If the available speedup cannot overcome the potential difference between best and worst case, then this approach may miss deadlines. If this is the case, MEANTIME sets a stricter latency goal than the user specified, forcing the controller to be more conservative. MEANTIME then aggressively sleeps or idles so that the user sees the desired latency.

3.3.4 Minimizing Accuracy Loss for Schedulability

The prior section derived the time to spend in the efficiency region for an application with a single approximate configuration. In practice, however, approximate computations expose a wide range of tradeoffs between accuracy and speedup [29, 53]. We denote the accuracy and speedup of application configuration i as a^i and s^i, respectively. To maximize the accuracy and maintain hard real-time constraints, we formulate the following opti-

[2] In practice, we can always idle or sleep if we reserve too much capacity and finish early.

mization problem:

$$maximize \sum_i \frac{t_s^i}{t} \cdot a^i \quad (9)$$

$$s.t.$$

$$\sum_i t_s^i \cdot r_{wc} \cdot s^i + t_e \cdot \Delta r_{wc} \geq W \quad (10)$$

$$t_{switch} + \sum_i t_s^i + t_e \leq t \quad (11)$$

$$t_s^i,\ t_e \geq 0 \quad (12)$$

Here, we assume that the user has supplied some set of approximations, all of which are acceptable, if not preferable. The goal is to find the maximum accuracy that guarantees the constraints in Eqns. 10–12. If this linear program has no feasible solution, the application is not schedulable on the system. MEANTIME, in general, does not need to solve this program often. If the acceptable accuracy never changes, then the program can be solved once at initialization time. If acceptable accuracy may change as the program runs, MEANTIME will solve a new program with a new set of variable accuracy configurations each time it changes.

Eqns. 10–12 have two non-trivial constraints. By the theory of linear programming, that means that there is an optimal solution with at most two t_s^i greater than 0 and all other equal to 0 [15]. Thus, during any interval, MEANTIME will switch configurations at most twice. This limitation on switching time means that we can provide a fairly tight bound on t_{switch}, meaning that switching time will have little impact on energy efficiency in practice.

3.3.5 Providing Feedback to the Controller

Control systems have been widely used to improve energy efficiency because control theory provides formal guarantees concerning how the system will react to dynamic events. For example, when an application shifts from a computationally intensive phase to an easy phase, the controller reacts to this change by reducing resource usage. The problem for hard real-time guarantees is that the control system reacts by detecting an error between the desired behavior and observed behavior. If the behavior under control is job latency, this error may result in missed deadlines.

This is an example of the fundamental tension between hard real-time and energy efficiency. We would like the runtime to reduce resource usage, but the control system will not detect the phase shift if every deadline is respected. MEANTIME overcomes this drawback by intercepting the latency feedback and altering it. Instead of providing the actual latency feedback (which will always be the same as all deadlines are guaranteed), MEANTIME estimates what the latency would have been if the application had not switched to a less accurate configuration and passes this latency to the controller. This latency estimate is denoted as $\hat{t}$ and calculated as:

$$\hat{t} = t_e + t_s \cdot s_0 + t_{switch} \quad (13)$$

Passing this estimated latency to the controller allows MEANTIME to maintain responsiveness and energy-efficiency while still meeting hard real-time deadlines.

Table 4: System configurations.

Configuration	Settings	Max Speedup	Max Powerup
big cores	4	4.52	2.00
big core speeds	19	10.23	10.42
LITTLE cores	4	4.52	1.32
LITTLE core speeds	13	7.11	2.62
idle	-	0	.6

3.4 MEANTIME Summary

MEANTIME works with existing control theoretic approaches that provide soft real-time support while minimizing energy consumption. MEANTIME augments these control based approaches in two novel ways. First, it computes the safety and efficiency regions as described above – thus combining hard real-time with energy efficiency. Second, it alters the feedback passed to the controller to maintain responsiveness to application phases despite the fact that deadlines are never violated.

4 Experimental Setup

4.1 Hardware Platform

Our hardware platform is an ODROID-XU3 from HardKernel. It runs Ubuntu Linux 14.04 using a modified kernel 3.10.58+. We use the `taskset` utility for managing processor core assignment and `cpufrequtils` for managing DVFS settings. This system supports five configurable resources that alter performance and power tradeoffs. Additionally, it has an extremely low-power *idle* state (about 0.1 Watt).

Table 4 summarizes system resources, showing the maximum increase in speed and power measured on the machine for any benchmark. Embedded INA-231 power sensors [34] provide power data for the big Cortex-A15 cluster, the LITTLE Cortex-A7 cluster, the DRAM and the GPU. The system sleeps at .1 Watts. The maximum measured power consumption is just under 6 Watts. *All energy and power numbers reported in this paper consider total usage as measured with the INA-321.*

4.2 Applications

We use six benchmark applications: x264, bodytrack, swaptions, ferret, and streamcluster (from the PARSEC

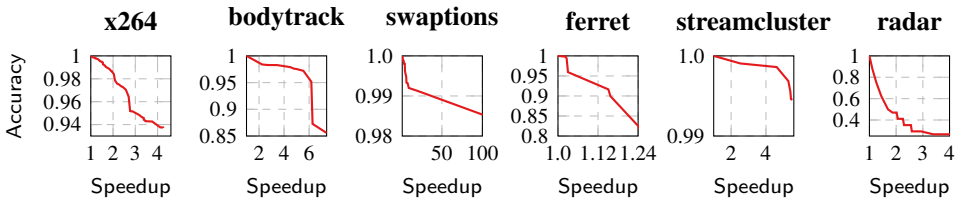


Figure 4: Speedup and accuracy tradeoffs for test applications.

Table 5: Approximate Application configurations.

App.	Configs.	Min. Spdup	Max Acc. Loss (%)
x264	560	3.96	6.2
bodytrack	200	5.24	14.4
swaptions	100	50.43	1.5
ferret	8	1.24	30.24
streamcluster	16	3.82	54.8
radar	512	3.95	73.4

Table 6: Application Input Details.

Application	Input	Jobs
x264	native	512 frames
bodytrack	sequenceB	261 frames
swaptions	randomized parameters	256 swaptions
ferret	corel	2000 queries
streamcluster	7 card poker hands	1000 hands
radar	radar pulses	100 pulses

benchmark suite [12]); as well as a radar processing application [28]. We use the PowerDial framework to modify all seven benchmarks so they support dynamic approximation [29]. PowerDial turns static configuration parameters into a data structure controlling the application's runtime behavior. PowerDial also instruments the applications, providing latency feedback for every outer loop iteration [29].

We measure application accuracy as a normalized distance from full accuracy. This is a standard metric allowing comparison across applications [49]. Changes in application performance are measured as speedup, the factor by which latency decreases when moving from the nominal setting. Unlike previous work on approximation that maximized average achievable speedup, MEANTIME cares about the worst case speedup. Thus, we measure the minimum speedup achievable with these transformations. This section describes the tradeoffs exposed by each of these applications. Table 5 summarizes the application-level configurations, showing the total number of available configurations as well as the minimum speedup and maximum accuracy loss. These applications represent a range of workloads which might be run on an embedded system with both timing and energy constraints. *None of these applications were originally intended to be run with hard-timing constraints making them a real test of the proposed technique.*

x264: This video encoder compresses a raw input according to the H.264 standard. It can decrease the frame latency at a cost of increased noise. x264 searches for redundancy both within a frame and between frames. The accuracy-aware x264 trades the work performed to find redundancy for the amount of redundancy identified. The total number of distinct application-level configurations is 560. The encoder's accuracy is measured by recording both the peak signal-to-noise ratio (PSNR) and the encoded bitrate, and weighting these two quantities equally.

bodytrack: This application uses an annealed particle filter to track a human moving through a space. The filter parameters trade the track's quality and the frame latency. The application exposes two knobs, one changes the number of annealing layers and another changes the number of particles. In total, the application supports 200 different configurations.

ferret: This application performs similarity search for images; *i.e.,* it finds images similar to a query image. Both the image analysis and the location sensitive hash used to find similar images support approximation. In total, ferret supports 8 different approximate configurations. The search accuracy is evaluated using F-measure, the harmonic mean of precision and recall[3].

streamcluster: This application is an online approximation of the k-means clustering algorithm. It has two parameters which trade clustering accuracy (measured using the B^3 score [2]) and clustering latency. The first changes the number of iterations used in the approximation, the second changes the distance metric used to assign a sample to a cluster. Together, they expose 16 different configurations.

swaptions: This financial analysis application uses Monte Carlo simulation to price a portfolio of swaptions. This application can reduce accuracy in the swaption price for decreased pricing latency. The application has a single parameter, with 100 settings, controlling the number of simulations per swaption.

radar: This application is the front-end of a radar signal processor and it turns raw antenna data into a target list. The application supports four different parameters that tradeoff signal-to-noise ratio (SNR) [4]. The first

[3]Precision is the number of returned documents relevant to a query divided by the total number of returned documents. Recall is the number of relevant documents returned divided by the total number of relevant documents.

[4]Higher SNR makes targets easier to detect, lower SNR makes tar-

Table 7: Application Timing Statistics.

	Latency Statistics (s)			
Application	Mean	Min	Max	STDEV/Mean
x264	1.33	0.14	2.97	0.59
bodytrack	0.75	0.64	0.92	0.11
swaptions	0.26	0.01	4.32	1.96
ferret	0.44	0.19	1.09	0.30
streamcluster	0.06	0.03	0.09	0.23
radar	0.03	0.03	0.05	0.03

two change the strength of the low-pass filter. The third changes the number of distinct directions the phased array antenna can "look." The fourth changes the range resolution. The radar can enter 512 separate configurations by changing these parameters.

All applications expose timing/accuracy tradeoffs. We illustrate the tradeoff spaces in Figure 4. Each chart's x-axis shows speedup and y-axis shows the resulting accuracy. Speedup and accuracy are both reported normalized to the default configuration. For clarity, we show only Pareto-optimal tradeoffs. Note that the y-axes show the range of accuracy tradeoffs for each application, and the x-axes show the range of possible speedups. Each application has a different range of both speedup and accuracy, so the axes also have different ranges.

4.3 Statistical Timing Characteristics

Table 6 shows the inputs for each benchmark. These benchmarks were not originally designed to provide predictable timing. We quantify this inherent unpredictability by measuring the latency of each job and computing the mean, minimum, maximum, and standard deviation over mean for all jobs in a benchmark. This data – summarized in Table 7 – shows that our benchmarks have a range of natural behavior from low variance (natural predictability; *e.g.,* bodytrack) to high variance (widely distributed job latencies; *e.g.,* swaptions).

This timing variability further motivates the need for MEANTIME, which meets hard real-time guarantees even for such off-the-shelf applications. The inherent variability means that allocating resources for worst case execution time requires reserving resources that are not used much of the time – the worst case is typically far from the average case. MEANTIME adapts to this variability by allocating for the average case to save energy and using a (typically) small amount of approximation to meet the hard real-time deadlines.

5 Experimental Evaluation

This section evaluates MEANTIME's ability to provide hard timing guarantees and energy savings. We first measure both latency and deadline misses. Next, we quantify MEANTIME's energy savings and then the effect of approximation. Next, we show how MEANTIME reacts to both changes in user goals (*e.g.,* transitioning to perfect accuracy) and application workload phases. We end by measuring MEANTIME's overhead.

We compare MEANTIME's timeliness, energy, and accuracy to the following approaches:

- **wcet**: meets hard real-time constraints by reserving resources sufficient for worst case latency and intelligently sleeping if a job finishes early [31]. We do not have a tool that can accurately estimate worst case latency on our test platform, so we use worst observed latency as a proxy.
- **PowerDial**: uses control theory to adjust application-level parameters and meet performance constraints with maximum accuracy [29]. PowerDial, however, cannot adjust system-level resource usage, so it cannot proactively lower system energy consumption. In fact, the only way to proactively achieve energy savings with PowerDial is to trade increased performance for reduced accuracy. The increased performance allows each job to finish earlier and then the system can spend more time in the idle state.
- **cross**: Extends existing cross-layer techniques (*e.g.,* [21, 58]) which combine application accuracy and system resource management. These prior works do not provide hard guarantees, but we extend them in a simple way by using worst case timing for both the application accuracy and system resource usage. This approach is much simpler than MEANTIME.
- **energy-aware**: uses state-of-the-art control and optimization techniques to allocate minimal energy resources for the current job [33, 42].
- **optimal**: is the true optimal energy-aware approach if we knew the future and the could instantly configure the system for each task. We compute the optimal by running each application in each system configuration and logging the latency of each job within the application. We then post-process these logs to determine the minimal energy configuration for each job that would have met the latency goal. This approach is obviously not practical, but we believe it represents the true energy savings available.

To test MEANTIME, we deploy it on our ODROID platform and run each application with a target latency equal to its maximum latency (as reported in Table 7). We measure missed deadlines and the average latency, energy, and accuracy for each job.

gets harder to detect. All configurations used in this paper still detect all targets with more than 10dB of margin.

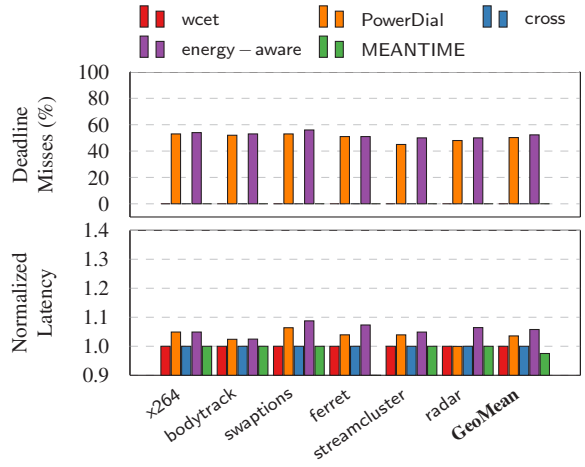


Figure 5: Deadline misses (top) and normalized latency (bottom) for different resource techniques.

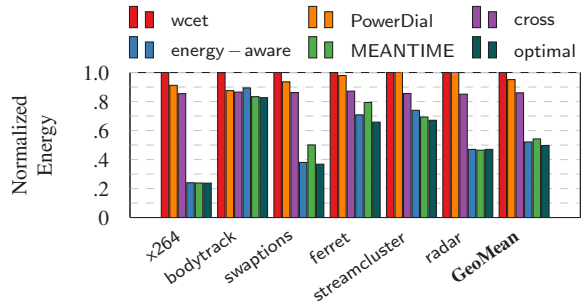


Figure 6: Energy consumption normalized to wcet. Lower numbers represent reduced energy consumption.

5.1 Timing Properties

Figure 5 shows the timing results with deadline misses on the top and latency (normalized to the target) on the bottom. These results confirm that none of wcet, cross, nor MEANTIME miss deadlines, thus they provide hard real-time guarantees. This agrees with the mathematical basis for MEANTIME established in Section 3, which should never miss deadlines as long as it is provided with accurate (or conservative) worst case timing information. PowerDial and energy-aware approach do not provide hard guarantees, so it is not surprising that they miss deadlines. Instead, these approaches support soft real-time goals, reflected by the average latency results in the bottom of Figure 5. Overall, these approaches provides good average timing; they are just not as predictable as wcet and MEANTIME.

5.2 Energy

Figure 6 shows each benchmark's energy consumption normalized to wcet. We normalize so that results are comparable across benchmarks. The results show that the energy-aware approach saves significant energy compared to wcet (confirming prior results [17, 32, 38]). By geometric mean, MEANTIME consumes only 54% of the energy of the wcet approach. MEANTIME's energy savings is comparable to the energy-aware approach, which consumes only 52% of wcet's energy, while the optimal approach consumes just 49% of wcet's energy.

The cross approach presented here consumes 86% of wcet's energy, which is a significant savings, but not close to MEANTIME. This cross layer approach is conservative in both system resource usage and application configuration, while MEANTIME is aggressive in resource allocation and conservative (for timing) in application configuration. The result show MEANTIME's combination achieves much better energy reductions. Similarly, PowerDial's energy savings is small – only about a 5% reduction – because PowerDial cannot proactivel alter system energy consumption. This result for PowerDial confirms previous work demonstrating that significant additional energy savings arises by coordinating application behavior with system resource usage [27].

Some benchmarks exhibit greater energy savings than others. In general, two factors predict the energy savings compared to wcet. First, the greater the variance in timing, the greater the energy saving potential – if an application spends most of its time near worst case latency, then there is limited opportunity to reduce resource usage. Second, if an application's approximate configurations do not provide much speedup, then the potential for energy savings is also reduced. Comparing the timing statistics (Table 7) and the speedups from approximation (Table 5) to the energy reduction in Figure 6 validates these observations. For instance, swaptions has much higher timing variance and speedups from approximation than bodytrack, and consequently, it exhibits better energy savings.

MEANTIME achieves large energy savings because allocating for worst case and idling after the job completes is one of the worst things to do on these processors. One study demonstrates that it is best to keep low-power multicores busy as much as possible [17]. Another shows that the Exynos Octa processor is more energy efficient at lower clockspeeds (and using the LITTLE cores) [32]. MEANTIME uses a control system in an attempt to keep the cores as busy as possible (and use the LITTLE cores as much as possible). On this architecture, the policy results in tremendous savings. On a another system where racing-to-idle is energy efficient, the MEANTIME approach would provide no benefit. We believe, however, that future embedded architectures will tend to look more like the processor used in this study and present a range of performance and energy tradeoffs.

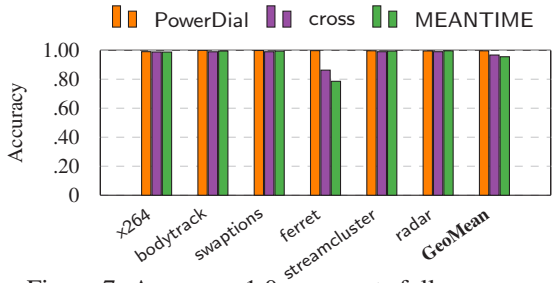


Figure 7: Accuracy, 1.0 represents full accuracy.

5.3 Accuracy

Figure 7 shows accuracy for PowerDial, cross, and MEANTIME. These results are normalized to nominal application behavior, so unity represents full accuracy. The results show generally high accuracy. Five of the six benchmarks have accuracies above 0.98 representing accuracy loss of less than 2%. The one exception is ferret, whose accuracy is 0.78. This is also the application for which the cross layer approach provides clear benefit.

While it may be hard to see, the accuracy for PowerDial is always higher than for MEANTIME. PowerDial really represents a technique for exposing and controlling performance/accuracy tradeoff spaces. In this case, we have configured PowerDial to produce the highest possible accuracy. An alternative could run in the lowest accuracy configuration to produce the highest possible performance, finish each task as early as possible, and spend the maximum time in the idle state. Doing so would produce the lowest accuracy output, but have significant energy savings. Prior work shows that the energy savings available in this case is always worse than coordinating between application and system [27].

Ferret's low accuracy is due to several factors. The first is the application's high variance (see Table 7). While this may at first appear beneficial, as it provides more room for high energy savings, adding the second factor creates problems. The second factor is the low speedup relative to the difference between minimum and maximum measured latency (see Table 5). The combination of high-variance (which is generally good for MEANTIME) and low speedup available at the application-level lead the application to spend much of its time in a reduced accuracy state. In contrast, because it is conservative in resource usage, cross spends less time in the reduced resource states that put more pressure on the application. MEANTIME's accuracy reduction results in fewer matches returned for a given search. The top (most relevant) matches are still returned, but less relevant ones are dropped. We emphasize that for users who do not want to lose this level of accuracy, they could simply specify fewer application knobs. The energy consumption would be higher, but that might be the tradeoff the user wants. The next section shows an example transitioning from reduced accuracy to perfect accuracy to support changing user goals.

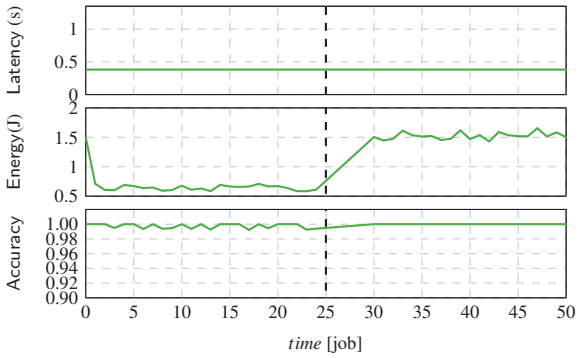


Figure 8: MEANTIME reacting to changing radar accuracy goals.

In general, for all approaches the accuracies are so high because very little speedup is needed most of the time; *i.e.,* most applications worst case behavior is far from the normal case so many jobs terminate in the safety zone. From our calculations of the true optimal energy savings available, we know the speedup required to meet the latency target for each job in each application. Table 8 shows the geometric mean of required speedup for each application across all its jobs. For most applications, the mean speedups required are quite small compared to the available speedups at the largest accuracy loss (compare Tables 5 and 8), so the accuracy losses are correspondingly small. The one exception is ferret, which has an achievable speedup of about 1.24×, but the mean required speedup is close to 1.14. This comparison helps explain ferret's lower accuracy.

Table 8: Required Speedup.

Application	Speedup
x264	1.18
bodytrack	1.06
swaptions	1.19
ferret	1.14
streamcluster	1.09
radar	1.03

5.4 Adapting to Changing Requirements

We transition a running application from energy minimization to accuracy maximization. To demonstrate this capability, we extend the radar example from Section 2. We now launch the radar application using MEANTIME to meet real-time goals and minimize energy consumption. After 25 jobs we change the radar's goal to real-time with maximum accuracy.

Figure 8 shows the results. The figure has three charts which show latency, energy, and accuracy as functions of time (measured in jobs). The vertical dashed lines represent the time when the acceptable approximation changes. The first half of the input is processed exactly as in the example section. The second half transitions

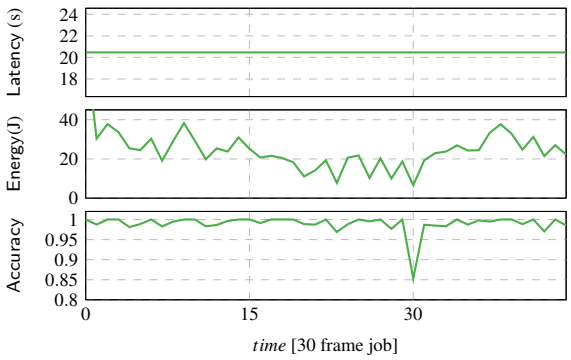


Figure 9: MEANTIME reacting to phases in x264.

to a higher energy configuration, but with no approximation. This demonstrates that the accuracy loss incurred by MEANTIME is optional and can be eliminated as needed. In this sense, MEANTIME represents a strictly greater set of capabilities than wcet.

5.5 Adapting to Phases

We run the x264 video encoder on an input consisting of three distinct video scenes (each of 450 frames). The difficulty (computational resources required to meet the target latency) varies from frame to frame, but on average, the first scene is the hardest, the second scene is the easiest, and the third scene is close to the first, but with a lower variance in frame-to-frame performance.

Figure 9 shows the results with latency, energy, and accuracy as a function of time (measured in 30 frame jobs). The latency target here is high, as it is hard to encode HD frames at high-quality on the test platform. MEANTIME achieves perfect timeliness, equivalent to wcet. These results demonstrate that MEANTIME provides predictable timing even as the application transitions through phases, something that prior energy-aware approaches which only support soft real-time cannot do [33]. Finally, these results show that the accuracy and energy are tailored to the application workload. During the middle scene (between jobs 15 and 29) the resource demand is lessened and the average accuracy improves slightly and the energy decreases. These results demonstrate that MEANTIME does not reduce accuracy needlessly, but tailors it to the workload requirements.

5.6 Overhead

MEANTIME's runtime is of constant – $O(1)$ – computational complexity. To make the system work in practice, however, it needs to know its own worst case latency so that it can ensure it does not interfere with the application's timing. To measure MEANTIME's latency, we deploy the runtime with no application to manage and "dummy resources" which are allocated through the same system calls, but do not change the timing. We execute 1000 iterations of the runtime and measure the worst case latency as approximately 100 μs. In practice, we account for this as part of the switching overhead as discussed in Section 3.

6 Related Work

The problem of scheduling for timeliness and energy efficiency has been widely studied in the literature. A complete survey is beyond the scope of this paper, but we mention some related highlights. At the coarsest level, scheduling and resource allocation can be done to provide either hard (*e.g.,* [3, 7, 9, 13, 20, 31, 37, 39, 47, 50, 52]) or soft (*e.g.,* [10, 11, 23, 25, 30, 33, 46]) timing guarantees, and in both cases it is beneficial to save as much energy as possible. While all these techniques differ in terms of the mechanisms they manipulate and the assumptions they make, we can draw some general conclusions. First, all techniques (whether hard or soft) manipulate *slack* – the time when the system is not busy. Some approaches scale voltage and frequency to reduce slack (*e.g.,* [13, 20, 25]); others manipulate processor sleep states (*e.g.,* [9, 31]). Still others work on general sets of resources specified at runtime (*e.g.,* [33, 44, 48, 54, 59]). For this paper, however, the important distinction is that soft guarantees allow the system to be much more aggressive about eliminating slack as they have the freedom to occasionally miss deadlines. Hard real-time guarantees, in contrast, require conservative allocation and never remove so much slack that timing might be violated [16]. It has even been noted that, in mixed criticality systems, aggressively removing slack for non-critical jobs can cause critical jobs to violate timing [57].

MEANTIME does not overturn these prior results. Rather, MEANTIME is based on the observation that we can achieve timing and energy efficiency if we make sacrifices in a third dimension. Specifically, MEANTIME exploits the growing domain of approximate computing. Approximate applications trade accuracy for performance, power, energy, or other benefits. Such approaches include both static, compile-time analysis of tradeoffs [5, 51, 53] and dynamic, runtime support for tradeoff management [4, 6, 18, 29, 49, 55]. Static analysis guarantees that accuracy bounds are never violated, but it is conservative and may miss chances for additional savings through dynamic customization.

Dynamic systems tailor behavior online. For example, Green maintains accuracy goals while minimizing energy [6], while Eon meets energy goals while maximizing accuracy [55]. Both Green and Eon use heuristic techniques for managing the tradeoff space. PowerDial [29], uses control theoretic techniques to provide performance guarantees while maximizing accu-

racy. PowerDial is the only technique that attempts to guarantee performance. Its control-theoretic guarantees may be appropriate for soft real-time, but it cannot provide hard real-time guarantees. None of these approaches combine hard real-time with energy reduction.

In contrast to these application-level approaches, many system-level designs reduce energy consumption by tailoring resource usage. To reduce total system power, most of these approaches coordinate multiple resources [19]. For example, Meisner et al. propose coordinating CPU power states, memories, and disks to meet soft latency goals while minimizing power consumption [43]. Bitirgen et al. coordinate clockspeed, cache, and memory bandwidth in a multicore [14]. Still other approaches focus on managing general sets of system-level components [33, 44, 48, 54, 59]. Finally, recent approaches manage system resources to provide both real-time and temperature guarantees, but do not minimize energy [22]. In fact, these systems provide either hard real-time guarantees (making no claims about energy), or they provide soft real-time guarantees (enforced through a variety of mechanisms) with energy savings.

Cross-layer optimization combines application-level approximation and system-level resource allocation. In that sense, MEANTIME most resembles other cross-layer approaches. Early cross-layer systems were designed for media processing on mobile systems. For example, Flinn and Satyanarayanan build a framework for coordinating operating systems and applications to meet user defined energy goals [21]. This system trades application quality for reduced energy consumption. GRACE [58] and GRACE-2 [56] use hierarchy to provide soft real-time guarantees for multimedia applications, making system-level adaptations first and then application-level adaptations. Like GRACE-2, Agilos uses hierarchy, combined with fuzzy control, to coordinate multimedia applications and systems to meet a performance goal [40]. Maggio et al. propose a game-theoretic approach for a decentralized coordination of application and system adaptation which provides soft real-time guarantees [41]. xTune uses static analysis to model application and system interaction and then refines that model with dynamic feedback [36]. CoAdapt allows users to pick two out of three of performance, power, and accuracy; it then provides soft guarantees in those two dimensions while optimizing the third [26].

Prior cross-layer approaches coordinate application accuracy and system energy, but none provide both hard real-time and energy minimization. Our empirical results show that a straightforward extension of existing cross layer approaches can meet hard real-time deadlines and saves energy compared to allocating for worst case and maximum accuracy, but it still consumes much more energy than MEANTIME (see Figure 6). *The combination of hard timing guarantees with the energy efficiency of an optimistic soft-timing system is the unique contribution of MEANTIME.*

7 Conclusion

This paper presents MEANTIME, a runtime control methodology. Its unique contribution is using application approximation to provide both hard real-time guarantees and energy efficiency. While not appropriate for all applications, MEANTIME provides a large benefit for those that can reduce accuracy. Our experimental results verify the claims made in the paper's introduction: MEANTIME achieves the timeliness of allocating for worst case and the energy efficiency of allocating for current case (actually, MEANTIME does slightly better). The capability provided by MEANTIME is strictly greater than allocating for worst case and racing to sleep. Therefore, we believe this is an important contribution, which can greatly increase energy efficiency for a class of embedded applications that must meet hard real-time constraints but can sacrifice a small amount of accuracy.

Acknowledgments The effort on this project is funded by the U.S. Government under the DARPA PERFECT program, the DARPA BRASS program, by the Dept. of Energy under DOE DE-AC02-06CH11357, by the NSF under CCF 1439156, and by a DOE Early Career Award. Additional funding for Anne Farrell comes from a GAANN fellowship.

References

[1] S. Albers. "Algorithms for Dynamic Speed Scaling". In: *STACS*. 2011, pp. 1–11.

[2] E. Amig, J. Gonzalo, J. Artiles, and F. Verdejo. "A comparison of extrinsic clustering evaluation metrics based on formal constraints". English. In: *Information Retrieval* 12.4 (2009), pp. 461–486. ISSN: 1386-4564. DOI: 10.1007/s10791-008-9066-8. URL: http://dx.doi.org/10.1007/s10791-008-9066-8.

[3] J. Anderson and S. Baruah. "Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms". In: *ICDCS*. 2004.

[4] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe. "Siblingrivalry: online autotuning through local competitions". In: *CASES*. 2012.

[5] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. "Language and compiler support for auto-tuning variable-accuracy algorithms". In: *CGO*. 2011.

[6] W. Baek and T. Chilimbi. "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation". In: *PLDI*. June 2010.

[7] M. Bambagini, M. Bertogna, and G. Buttazzo. "On the effectiveness of energy-aware real-time scheduling algorithms on single-core platforms". In: *ETFA*. 2014.

[8] N. Bansal, D. P. Bunde, H.-L. Chan, and K. Pruhs. "Average Rate Speed Scaling". In: *Algorithmica* 60.4 (2011).

[9] P. Baptiste. "Scheduling Unit Tasks to Minimize the Number of Idle Periods: A Polynomial Time Algorithm for Offline Dynamic Power Management". In: *SODA*. 2006.

[10] M. Becker, A. Schmidt, M. Orehek, and T. Nolte. "Saving energy by means of dynamic load management in embedded multicore systems". In: *SIES*. 2014.

[11] L. Bertini, J. Leite, and D. Mosse. "Statistical QoS Guarantee and Energy-Efficiency in Web Server Clusters". In: *ECRTS*. 2007.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li. "The PARSEC Benchmark Suite: Characterization and Architectural Implications". In: *PACT*. 2008.

[13] E. Bini, G. Buttazzo, and G. Lipari. "Minimizing CPU Energy in Real-time Systems with Discrete Speed Management". In: *ACM Trans. Embed. Comput. Syst.* 8.4 (July 2009), 31:1–31:23.

[14] R. Bitirgen, E. Ipek, and J. F. Martinez. "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach". In: *MICRO*. 2008.

[15] S. Bradley, A. Hax, and T. Magnanti. *Applied mathematical programming*. 1977.

[16] G. C. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency: Predictability vs. Efficiency*. Springer, 2006.

[17] A. Carroll and G. Heiser. "Mobile multicores: use them or waste them". In: *Operating Systems Review* 48.1 (2014), pp. 44–48. DOI: 10.1145/2626401.2626411. URL: http://doi.acm.org/10.1145/2626401.2626411.

[18] F. Chang and V. Karamcheti. "Automatic Configuration and Run-time Adaptation of Distributed Applications". In: *HPDC*. 2000.

[19] H. Cheng and S. Goddard. "SYS-EDF: a system-wide energy-efficient scheduling algorithm for hard real-time systems". In: *International Journal of Embedded Systems* 4.2 (2009).

[20] A. Dudani, F. Mueller, and Y. Zhu. "Energy-conserving Feedback EDF Scheduling for Embedded Systems with Real-time Constraints". In: *LCTES/SCOPES '02*. 2002.

[21] J. Flinn and M. Satyanarayanan. "Managing battery lifetime with energy-aware adaptation". In: *ACM Trans. Comp. Syst.* 22.2 (May 2004).

[22] Y. Fu, N. Kottenstette, C. Lu, and X. D. Koutsoukos. "Feedback thermal control of real-time systems on multicore processors". In: *EMSOFT*. 2012.

[23] R. Guerra, J. C. B. Leite, and G. Fohler. "Attaining soft real-time constraint and energy-efficiency in web servers". In: *SAC*. 2008.

[24] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. ISBN: 047126637X.

[25] J. Heo, P. Jayachandran, I. Shin, D. Wang, T. Abdelzaher, and X. Liu. "OptiTuner: On Performance Composition and Server Farm Energy Minimization Application". In: *IEEE Transactions on Parallel and Distributed Systems* 22.11 (2011).

[26] H. Hoffmann. “CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems”. In: *ECRTS*. 2014.

[27] H. Hoffmann. “JouleGuard: Energy Guarantees for Approximate Applications”. In: *SOSP*. 2015.

[28] H. Hoffmann, A. Agarwal, and S. Devadas. “Selecting Spatiotemporal Patterns for Development of Parallel Applications”. In: *IEEE Trans. Parallel Distrib. Syst.* 23.10 (2012), pp. 1970–1982. DOI: 10.1109/TPDS.2011.298. URL: http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.298.

[29] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. “Dynamic Knobs for Responsive Power-Aware Computing”. In: *ASPLOS*. 2011.

[30] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. “Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control”. In: *Computers, IEEE Transactions on* 56.4 (2007), pp. 444 –458.

[31] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. Buttazzo. “Adaptive Dynamic Power Management for Hard Real-Time Systems”. In: *RTSS*. 2009.

[32] C. Imes and H. Hoffmann. “Minimizing energy under performance constraints on embedded platforms: resource allocation heuristics for homogeneous and single-ISA heterogeneous multi-cores”. In: *SIGBED Review* 11.4 (2014), pp. 49–54. DOI: 10.1145/2724942.2724950. URL: http://doi.acm.org/10.1145/2724942.2724950.

[33] C. Imes, D. H. K. Kim, M. Maggio, and H. Hoffmann. “POET: A Portable Approach to Minimizing Energy Under Soft Real-time Constraints”. In: *RTAS*. 2015.

[34] T. Instruments. http://www.ti.com/product/ina231.

[35] S. Irani, S. Shukla, and R. Gupta. “Algorithms for Power Savings”. In: *ACM Trans. Algorithms* 3.4 (Nov. 2007).

[36] M. Kim, M.-O. Stehr, C. Talcott, N. Dutt, and N. Venkatasubramanian. “xTune: A Formal Methodology for Cross-layer Tuning of Mobile Embedded Systems”. In: *ACM Trans. Embed. Comput. Syst.* 11.4 (Jan. 2013).

[37] F. Kong, Y. Wang, Q. Deng, and W. Yi. “Minimizing Multi-resource Energy for Real-Time Systems with Discrete Operation Modes”. In: *ECRTS*. 2010.

[38] E. Le Sueur and G. Heiser. “Slow Down or Sleep, That is the Question”. In: *USENIX ATC*. 2011. URL: http://dl.acm.org/citation.cfm?id=2002181.2002197.

[39] Y.-H. Lee, K. Reddy, and C. Krishna. “Scheduling techniques for reducing leakage power in hard real-time systems”. In: *ECRTS*. 2003.

[40] B. Li and K. Nahrstedt. “A control-based middleware framework for quality-of-service adaptations”. In: *IEEE Journal on Selected Areas in Communications* 17.9 (Sept. 1999).

[41] M. Maggio, E. Bini, G. C. Chasparis, and K.-E. Årzén. “A Game-Theoretic Resource Manager for RT Applications”. In: *ECRTS*. 2013.

[42] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. “Power Optimization in Embedded Systems via Feedback Control of Resource Allocation”. In: *IEEE Trans. on Control Systems Technology* 21.1 (2013).

[43] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. “Power management of online data-intensive services”. In: *ISCA* (2011).

[44] N. Mishra, H. Zhang, J. D. Lafferty, and H. Hoffmann. “A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints”. In: *ASPLOS*. 2015.

[45] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar. “Critical Power Slope: Understanding the Runtime Effects of Frequency Scaling”. In: *ICS*. 2002. ISBN: 1-58113-483-5. DOI: 10.1145/514191.514200. URL: http://doi.acm.org/10.1145/514191.514200.

[46] R. Nassiffe, E. Camponogara, and G. Lima. “Optimizing QoS in Energy-aware Real-time Systems”. In: *SIGBED Rev.* 10.2 (July 2013).

[47] R. Racu, A. Hamann, R. Ernst, B. Mochocki, and X. S. Hu. “Methods for power optimization in distributed embedded systems with real-time requirements”. In: *CASES*. 2006.

[48] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. “A resource allocation model for QoS management”. In: *RTSS*. 1997.

[49] M. Rinard. “Probabilistic accuracy bounds for fault-tolerant computations that discard tasks”. In: *ICS*. 2006.

[50] S. Saha, J. S. Deogun, and Y. Lu. “Adaptive energy-efficient task partitioning for heterogeneous multi-core multiprocessor real-time systems”. In: *HPCS*. 2012.

[51] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. “EnerJ: approximate data types for safe and general low-power computation”. In: *PLDI*. 2011.

[52] A. Shrivastava, E. Earlie, N. Dutt, and A. Nicolau. “Aggregating Processor Free Time for Energy Reduction”. In: *CODES+ISSS*. 2005.

[53] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. “Managing performance vs. accuracy trade-offs with loop perforation ”. In: *ESEC/FSE*. 2011.

[54] M. Sojka, P. Písa, D. Faggioli, T. Cucinotta, F. Checconi, Z. Hanzálek, and G. Lipari. “Modular software architecture for flexible reservation mechanisms on heterogeneous resources”. In: *Journal of Systems Architecture* 57.4 (2011).

[55] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. “Eon: a language and runtime system for perpetual systems”. In: *SenSys*. 2007.

[56] V. Vardhan, W. Yuan, A. F. H. III, S. V. Adve, R. Kravets, K. Nahrstedt, D. G. Sachs, and D. L. Jones. “GRACE-2: integrating fine-grained application adaptation with global adaptation for saving energy”. In: *IJES* 4.2 (2009).

[57] M. Volp, M. Hahnel, and A. Lackorzynski. “Has energy surpassed timeliness? Scheduling energy-constrained mixed-criticality systems”. In: *RTAS*. 2014.

[58] W. Yuan and K. Nahrstedt. “Energy-efficient soft real-time CPU scheduling for mobile multimedia systems”. In: *ACM SIGOPS Operating Systems Review* 37.5 (2003), pp. 149–163.

[59] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. “ControlWare: A middleware architecture for Feedback Control of Software Performance”. In: *ICDCS*. 2002.

Design Guidelines for High Performance RDMA Systems

Anuj Kalia Michael Kaminsky[†] David G. Andersen
Carnegie Mellon University *[†]Intel Labs*

Abstract

Modern RDMA hardware offers the potential for exceptional performance, but design choices including *which* RDMA operations to use and *how* to use them significantly affect observed performance. This paper lays out guidelines that can be used by system designers to navigate the RDMA design space. Our guidelines emphasize paying attention to low-level details such as individual PCIe transactions and NIC architecture. We empirically demonstrate how these guidelines can be used to improve the performance of RDMA-based systems: we design a networked sequencer that outperforms an existing design by 50x, and improve the CPU efficiency of a prior high-performance key-value store by 83%. We also present and evaluate several new RDMA optimizations and pitfalls, and discuss how they affect the design of RDMA systems.

1 Introduction

In recent years, Remote Direct Memory Access (RDMA)-capable networks have dropped in price and made substantial inroads into datacenters. Despite their newfound popularity, using their advanced capabilities to best effect remains challenging for software designers. This challenge arises because of the nearly-bewildering array of options a programmer has for using the NIC[1], and because the relative performance of these operations is determined by complex low-level factors such as PCIe bus transactions and the (proprietary and often confidential) details of the NIC architecture.

Unfortunately, finding an efficient match between RDMA capabilities and an application is important: As we show in Section 5, the best and worst choices of RDMA options vary by a factor of *seventy* in their overall throughput, and by a factor of 3.2 in the amount of host CPU they consume. Furthermore, there is no one-size-fits-all best approach. Small changes in application requirements significantly affect the relative performance of different designs. For example, using general-purpose RPCs over RDMA is the best design for a networked key-value store (Section 4), but this same design choice provides lower scalability and 16% lower throughput than the best choice for a networked "sequencer" (Section 4; the sequencer returns a monotonically increasing integer to client requests).

This paper helps system designers address this challenge in two ways. First, it provides guidelines, backed by an open-source set of measurement tools, for evaluating and optimizing the most important system factors that affect end-to-end throughput when using RDMA NICs. For each guideline (e.g., "Avoid NIC cache misses"), the paper provides insight on both how to determine whether this guideline is relevant (e.g., by using PCIe counter measurements to detect excess traffic between the NIC and the CPU), and a discussion of which modes of using the NICs are most likely to mitigate the problem.

Second, we evaluate the efficacy of these guidelines by applying them to both microbenchmarks and real systems, across three generations of RDMA hardware. Section 4.2 presents a novel design for a network sequencer that outperforms an existing design by 50x. Our best sequencer design handles 122 million operations/second using a single NIC and scales well. Section 4.3 applies the guidelines to improve the CPU efficiency and throughput of the HERD key-value cache [20] by 83% and 35% respectively. Finally, we show that today's RDMA NICs handle contention for atomic operations extremely slowly, rendering designs that use them [27, 30, 11] very slow.

A lesson from our work is that low-level details are surprisingly important for RDMA system design. Our underlying goal is to provide researchers and developers with a roadmap through these details without necessarily becoming RDMA gurus. We provide simple models of RDMA operations and their associated CPU and PCIe costs, plus open-source software to measure and analyze them (`https://github.com/efficient/rdma_bench`).

We begin our journey into high-performance RDMA-based systems with a review of the relevant capabilities of RDMA NICs, and the PCIe bus that frequently arises as a bottleneck.

[1]In this paper, we refer exclusively to RDMA-capable network interface cards, so we use the more generic but shorter term NIC throughout.

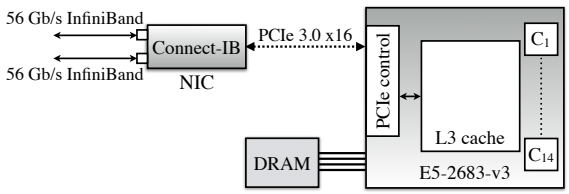


Figure 1: Hardware components of a node in an RDMA cluster

2 Background

Figure 1 shows the relevant hardware components of a machine in an RDMA cluster. A NIC with one or more ports connects to the PCIe controller of a multi-core CPU. The PCIe controller reads/writes the L3 cache to service the NIC's PCIe requests; on modern Intel servers [4], the L3 cache provides counters for PCIe events.

2.1 PCI Express

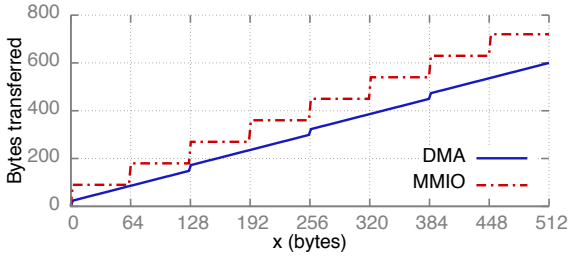
The current fastest PCIe link is PCIe "3.0 x16," the 3rd generation PCIe protocol, using 16 lanes. The bandwidth of a PCIe link is the per-lane bandwidth times the number of lanes. PCIe is a layered protocol, and the layer headers add overhead that is important to understand for efficiency. RDMA operations generate 3 types of PCIe transaction layer packets (TLPs): read requests, write requests, and read completions (there is no transaction-layer response for a write). Figure 2a lists the bandwidth and header overhead for the PCIe generations in our clusters. Note that the header overhead of 20–26 bytes is comparable to the common size of data items used in services such as memcached [25] and RPCs [15].

MMIO writes vs. DMA reads There are important differences between the two methods of transferring data from a CPU to a PCIe device. CPUs write to mapped device memory (MMIO) to initiate PCIe writes. To avoid generating a PCIe write for each store instruction, CPUs use an optimization called "write combining," which combines stores to generate cache line–sized PCIe transactions. PCIe devices have DMA engines and can read from DRAM using DMA. DMA reads are not restricted to cache lines, but a read response larger than the CPU's read completion combining size (C_{rc}) is split into multiple completions. C_{rc} is 128 bytes for the Intel CPUs used in our measurements (Table 2); we assume 128 bytes for the AMD CPU [4, 3]. A DMA read always uses less host-to-device PCIe bandwidth than an equal-sized MMIO; Figure 2b shows an analytical comparison. This is an important factor, and we show how it affects performance of higher-layer protocols in the subsequent sections.

PCIe counters Our contributions rely on understanding the PCIe interaction between NICs and CPUs. Although precise PCIe analysis requires expensive PCIe analyzers or proprietary/confidential NICs manuals, PCIe counters available on modern CPUs can provide several useful insights.[2] For each counter, the number of captured events per second is its *counter rate*. Our analysis primarily uses counters for DMA reads (`PCIeRdCur`) and DMA writes (`PCIeItoM`).

Gen	Bitrate	Per-lane b/w	Request	Completion
2.0	5 GT/s	500 MB/s	24 B	20 B
3.0	8 GT/s	984.6 MB/s	26 B	22 B

(a) *Speed and header sizes for PCIe generations. Lane bandwidth excludes physical layer encoding overhead.*



(b) *CPU-to-device PCIe traffic for an x-byte transfer with DMA and MMIO, assuming PCIe 3.0 and C_{rc} = 128 bytes.*

Figure 2: PCIe background

2.2 RDMA

RDMA is a network feature that allows direct access to the memory of a remote computer. RDMA-providing networks include InfiniBand, RoCE (RDMA over Converged Ethernet), and iWARP (Internet Wide Area RDMA Protocol). RDMA networks usually provide high bandwidth and low latency: NICs with 100 Gbps of per-port bandwidth and ~ 2µs round-trip latency are commercially available. The performance and scalability of an RDMA-based communication protocol depends on several factors including the operation (verb) type, transport, optimization flags, and operation initiation method.

2.2.1 RDMA verbs and transports

RDMA hosts communicate using queue pairs (QPs); hosts create QPs consisting of a send queue and a receive queue, and post operations to these queues using the *verbs* API. We call the host initiating a verb the *requester* and the destination host the *responder*. For some verbs, the responder does not actually send a response. On completing a verb, the requester's NIC optionally signals completion by DMA-ing a completion entry (CQE) to a completion queue (CQ) associated with the QP. Verbs can be made *unsignaled* by setting a flag in the request; these verbs do not generate a CQE, and the application detects completion using application-specific methods.

The two types of verbs are memory verbs and messaging verbs. Memory verbs include RDMA reads, writes,

[2] The CPU intercepts cache line-level activity between the PCIe controller and the L3 cache, so the counters can miss some critical information. For example, the counters indicate 2 PCIe reads when the NIC reads a 4-byte chunk straddling 2 cache lines.

	SEND/RECV	WRITE	READ	WQE header
RC	✓	✓	✓	36 B
UC	✓	✓	✗	36 B
UD	✓	✗	✗	68 B

Table 1: Operations supported by each transport type, and their Mellanox WQE header size for SEND, WRITE, and READ. RECV WQE header size is 16 B for all transports.

and atomic operations. These verbs specify the remote address to operate on and bypass the responder's CPU. Messaging verbs include the send and receive verbs. These verbs involve the responder's CPU: the send's payload is written to an address specified by a receive that was posted previously by the responder's CPU. In this paper, we refer to RDMA read, write, send, and receive verbs as READ, WRITE, SEND, and RECV respectively.

RDMA transports are either reliable or unreliable, and either connected or unconnected (also called datagram). With reliable transports, the NIC uses acknowledgments to guarantee in-order delivery of messages. Unreliable transports do not provide this guarantee. However, modern RDMA implementations such as InfiniBand use a lossless link layer that prevents congestion-based losses using link layer flow control [1], and bit error–based losses using link layer retransmissions [8]. Therefore, unreliable transports drop packets very rarely. Connected transports require one-to-one connections between QPs, whereas a datagram QP can communicate with multiple QPs. We consider two types of connected transports in this paper: Reliable Connected (RC) and Unreliable Connected (UC). Current RDMA hardware provides only 1 datagram transport: Unreliable Datagram (UD). Different transports support different subsets of verbs: UC does not support RDMA reads, and UD does not support memory verbs. Table 1 summarizes this.

2.2.2 RDMA WQEs

To initiate RDMA operations, the user-mode NIC driver at the requester creates Work Queue Elements (WQEs) in host memory; typically, WQEs are created in a pre-allocated, contiguous memory region, and each WQE is individually cache line–aligned. (We discuss methods of transferring WQEs to the device in Section 3.1.) The WQE format is vendor-specific and is determined by the NIC hardware.

WQE size depends on several factors: the type of RDMA operation, the transport, and whether the payload is referenced by a pointer field or *inlined* in the WQE (i.e., the WQE buffer includes the payload). Table 1 shows the WQE header size for Mellanox NICs for three transports. For example, with a 36-byte WQE header, the size of a WRITE WQE with an x-byte inlined payload is $36 + x$ bytes. UD WQEs have larger, 68-byte headers to store additional routing information.

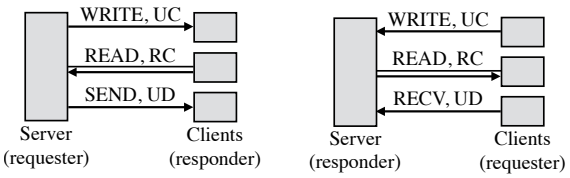


Figure 3: Inbound and outbound verbs at the server.

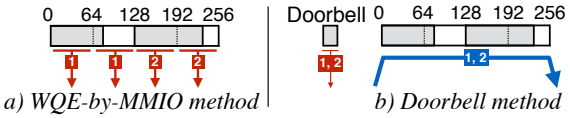


Figure 4: The WQE-by-MMIO and Doorbell methods for transferring two WQEs (shaded) spanning 2 cache lines. Arrows represent PCIe transactions. Red (thin) arrows are MMIO writes; the blue (thick) arrow is a DMA reads. Arrows are marked with WQE numbers; arrow width represents transaction size.

2.2.3 Terminology and default assumptions

We distinguish between *inbound* and *outbound* verbs because their performance differs significantly (Section 5): memory verbs and SENDs are outbound at the requester and inbound at the responder; RECVs are always inbound. Figure 3 summarizes this. As our study focuses on small messages, all WRITEs and SENDs are inlined by default. We define the padding-to-cache-line-alignment function $x' := \lceil x/64 \rceil * 64$. We denote the per-lane bandwidth, request header size, and completion header size of PCIe 3.0 by P_{bw}, P_r, and P_c, respectively.

3 RDMA design guidelines

We now present our design guidelines. Along with each guideline, we present new optimizations and briefly describe those presented in prior literature. We make two assumptions about the NIC hardware that are true for all currently available NICs. First, we assume that the NIC is PCIe-based device. Current network interfaces (NIs) are predominantly discrete PCIe cards; vendors are beginning to integrate NIs on-die [2, 5], or on-package [6], but these NIs still communicate with the PCIe controller using the PCIe protocol, and are less powerful than discrete NIs. Second, we assume that the NIC has internal parallelism using multiple processing units (PUs)—this is generally true of high-speed NIs [19]. As in conventional parallel programming, this parallelism provides both optimization opportunities (Section 3.3) and pitfalls (Section 3.4).

To discuss the impact on CPU and PCIe use of the optimizations below, we consider transferring N WQEs of size D bytes from the CPU to the NIC.

3.1 Reduce CPU-initiated MMIOs

Both CPU efficiency and RDMA throughput can improve if MMIOs are reduced or replaced with the more CPU-

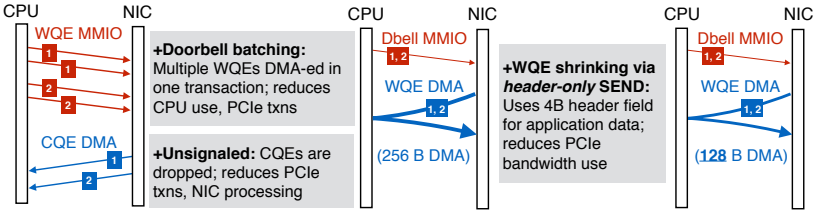


Figure 5: Optimizations for issuing two 4-byte UD SENDs. A UD SEND WQE spans 2 cache lines on Mellanox NICs because of the 68-byte header; we shrink it to 1 cache line by using a 4-byte header field for payload. Arrow notation follows Figure 4.

and bandwidth-efficient DMAs. CPUs initiate network operations by sending a message to the NIC via MMIO. The message can (1) contain the new work queue elements, or (2) it can refer to the new WQEs by using information such as the address of the last WQE. In the first case, the WQEs are transferred via 64-byte write-combined MMIOs. In the second case, the NIC reads the WQEs using one or more DMAs.[3] We refer to these methods as *WQE-by-MMIO* and *Doorbell* respectively. (Different technologies have different terms for these methods. Mellanox uses "BlueFlame" and "Doorbell," and Intel® Omni-Path Architecture uses "PIO send" and "SDMA," respectively.) Figure 4 summarizes this. The WQE-by-MMIO method optimizes for low latency and is typically the default. Two optimizations can improve performance by reducing MMIOs:

Doorbell batching If an application can issue multiple WQEs to a QP, it can use one Doorbell MMIO for the batch. *CPU:* Doorbell reduces CPU-generated MMIOs from $N * D' / 64$ with WQE-by-MMIO to 1. *PCIe:* For $N = 10$ and $D = 65$ and PCIe 3.0, Doorbell transfers 1534 bytes, whereas WQE-by-MMIO transfers 1800 bytes (Appendix A).

In this paper, we refer to Doorbell batching as batching—a batched WQE transfer via WQE-by-MMIO is identical to a sequence of individual WQE-by-MMIOs, so batching is only useful for Doorbell.

WQE shrinking Reducing the number of cache lines used by a WQE can improve throughput drastically. For example, consider reducing WQE size by only 1 byte from 129 B to 128 B, and assume that WQE-by-MMIO is used. *CPU:* CPU-generated MMIOs decreases from 3 to 2. *PCIe:* Number of PCIe transactions decreases from 3 to 2. Shrinking mechanisms include compacting the application payload, or overloading unused WQE header fields with application data.

3.2 Reduce NIC-initiated DMAs

Reducing DMAs saves NIC processing power and PCIe bandwidth, improving RDMA throughput. Note that the batching optimization above *adds* a DMA read, but it avoids *multiple* MMIOs, which is usually a good tradeoff.

[3]In this paper, we assume that the NIC reads *all* new WQEs in one DMA, as is done by Mellanox's Connect-IB and newer NICs. Older Mellanox NICs read one or more WQEs per DMA, depending on the NIC's proprietary prefetching logic.

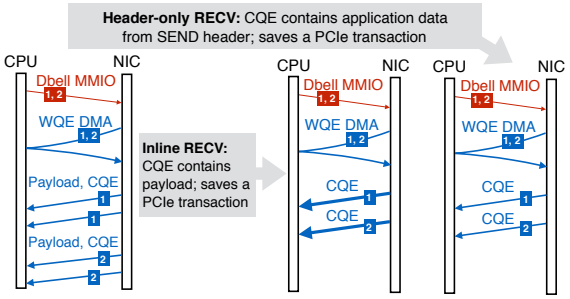


Figure 6: Optimizations for RECVs with small SENDs.

Known optimizations to reduce NIC-initiated DMAs include unsignaled verbs which avoid the completion DMA write (Section 2.2.1), and payload inlining which avoids the payload DMA read (Section 2.2.2). The two optimizations in Section 3.1 affect DMAs, too: batching with large N requires fewer DMA reads than smaller N; WQE shrinking further makes these reads smaller.

NICs must DMA a completion queue entry for completed RECVs [1]; this provides an additional optimization opportunity, as discussed below. Unlike CQEs of other verbs that only signal completion and are dispensible, RECV CQEs contain important metadata such as the size of received data. NICs typically generate two separate DMAs for payload and completion, writing them to application- and driver-owned memory respectively. We later show that the corresponding performance penalty explains the rule-of-thumb that messaging verbs are slower than memory verbs, and using the DMA-avoiding optimizations below challenges this rule-of-thumb for some payload sizes. We assume here that the corresponding SEND for a RECV carries an X-byte payload.

Inline RECV If X is small (~ 64 for Mellanox's NICs), the NIC encapsulates the payload in the CQE, which is later copied by the driver to the application-specified address. *CPU:* Minor overhead for copying the small payload. *PCIe:* Uses 1 DMA instead of 2.

Header-only RECV If $X = 0$ (i.e., the RDMA SEND packet consists of only a header and no payload), the payload DMA is not generated at the receiver. Some information from the packet's header is included in the DMA-ed CQE, which can be used to implement application protocols. We call SENDs and RECVs with $X = 0$ *header-only*, and *regular* otherwise. *PCIe:* Uses 1 DMA instead of 2.

Figure 5 and Figure 6 summarize these two guidelines for UD SENDs and RECVs, respectively. These two verbs are used extensively in our evaluation.

3.3 Engage multiple NIC PUs

Exploiting NIC parallelism is necessary for high performance, but requires explicit attention. A common RDMA programming decision is to use as few queue pairs as possible, but doing so limits NIC parallelism to the number of QPs. This is because operations on the same QP have ordering dependencies and are ideally handled by the same NIC processing unit to avoid cross-PU synchronization. For example, in datagram-based communication, one QP per CPU core is sufficient for communication with all remote cores. Using one QP consumes the least NIC SRAM to hold QP state, while avoiding QP sharing among CPU cores. However, it "binds" a CPU core to a PU and may limit core throughput to PU throughput. This is likely to happen when per-message application processing is small (e.g., the sequencer in Section 4.2) and a high-speed CPU core overwhelms a less powerful PU. In such cases, using multiple QPs per core increases CPU efficiency; we call this the *multi-queue* optimization.

3.4 Avoid contention among NIC PUs

RDMA operations that require cross-QP synchronization introduce contention among PUs, and can perform over an order of magnitude worse than uncontended operations. For example, RDMA provides atomic operations such as compare-and-swap and fetch-and-add on remote memory. To our knowledge, all NICs available at the time of writing (including the recently released ConnectX-4 [7]) use internal concurrency control for atomics: PUs acquire an internal lock for the target address and issue read-modify-write over PCIe. Note that atomic operations contend with non-atomic verbs too. Future NICs may use PCIe's atomic transactions for higher performing, cache coherence-based concurrency control.

Therefore, the NIC's internal locking *mechanism*, such as the number of locks and the mapping of atomic addresses to these locks, is important; we describe experiments to infer this in Section 5.4. Note that due to the limited SRAM in NICs, the number of available locks is small, which amplifies contention in the workload.

3.5 Avoid NIC cache misses

NICs cache several types of information; it is critical to maintain a high cache hit rate because a miss translates to a read over PCIe. Cached information includes (1) virtual to physical address translations for RDMA-registered memory, (2) QP state, and (3) a work queue element cache. While the first two are known [13], the third is undocumented and was discovered in our experiments. Address translation cache misses can be reduced by using large (e.g., 2 MB) pages, and QP state cache misses by using fewer QPs [13]. We make two new contributions in this context:

Name	Hardware
CX	ConnectX (1x 20 Gb/s InfiniBand ports), PCIe 2.0 x8, AMD Opteron 8354 (4 cores, 2.2 GHz)
CX3	ConnectX-3 (1x 56 Gb/s InfiniBand ports), PCIe 3.0 x8, Intel® Xeon® E5-2450 CPU (8 cores, 2.1 GHz)
CIB	Connect-IB (2x 56 Gb/s InfiniBand ports), PCIe 3.0 x16, Intel® Xeon® E5-2683-v3 CPU (14 cores, 2 GHz)

Table 2: Measurement clusters. CX is NSF PRObE's Nome cluster [17], CX3 is Emulab's Apt cluster [31], and CIB is a cluster at NetApp. CX uses PCIe 2.0 at 2.5 GT/s.

Detecting cache misses All types of NIC cache misses are transparent to the application and can be difficult to detect. We demonstrate how PCIe counters can be leveraged to accomplish this, by detecting and measuring WQE cache misses (Section 5.3.2). In general, subtracting the application's expected PCIe reads from the actual reads reported by PCIe counters gives an estimate of cache misses. Estimating expected PCIe reads in turn requires PCIe models of RDMA operations (Section 5.1).

WQE cache misses The initial work queue element transfer from CPU to NIC triggers an insertion of the WQE into the NIC's WQE cache. When the NIC eventually processes this WQE, a cache miss can occur if it was evicted by newer WQEs. In Section 5.3.2, we show how to measure and reduce these misses.

4 Improved system designs

We now demonstrate how these guidelines can be used to improve the design of whole systems. We consider two systems: networked sequencers, and key-value stores.

Evaluation setup We perform our evaluation on the three clusters described in Table 2. We name the clusters with the initials of their NICs, which is the main hardware component governing performance. CX3 and CIB run Ubuntu 14.04 with Mellanox OFED 2.4; CX runs Ubuntu 12.04 with Mellanox OFED 2.3. Throughout the paper, we use WQE-by-MMIO for non-batched operations and Doorbell for batched operations. However, when batching is enabled but the available batch size is one, WQE-by-MMIO is used. (Doorbell provides little CPU savings for transferring a single small WQE, and uses an extra PCIe transaction.) For brevity, we primarily use the state-of-the-art CIB cluster in this section; Section 5 evaluates our optimizations on all clusters.

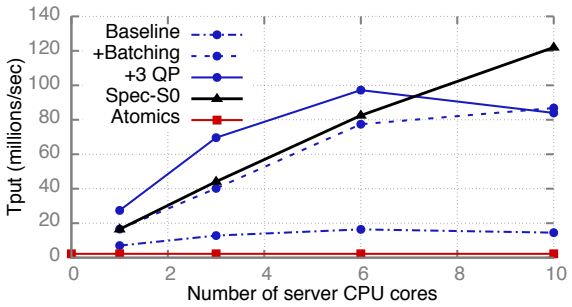


Figure 7: Impact of optimizations on HERD RPC-based sequencer (blue lines with circular dots), and throughput of `Spec-S0` and the atomics-based sequencer

4.1 Overview of HERD RPCs

We use HERD's RPC protocol for communication between clients and the sequencer/key-value server. HERD RPCs have low overhead at the server and high number-of-clients scalability. Protocol clients use unreliable WRITEs to write requests to a request memory region at the server. Before doing so, they post a RECV to an unreliable datagram QP for the server's response. A server thread (a *worker*) detects a new request by polling on the request memory region. Then, it performs application processing and responds using a UD SEND posted via WQE-by-MMIO.

We apply the following two optimizations to HERD RPCs in general; we present specific optimizations for each system later.

- **Batching** Instead of responding after detecting one request, the worker checks for one request from each of the C clients, collecting $N \leq C$ requests. Then, it SENDs N responses using a batched Doorbell.
- **Multi-queue** Each worker alternates among a tuneable number of UD queue pairs across the batched SENDs.

Note that batching does not add significant latency because we do it opportunistically [23, 21]; we do not wait for a number of requests to accumulate. We briefly discuss the latency added by batching in Section 4.2.2.

4.2 Networked sequencers

Centralized sequencers are useful building blocks for a variety of network applications, such as ordering operations in distributed systems via logical or real timestamps [11], or providing increasing offsets into a linearly growing memory region [10]. A centralized sequencer can be the bottleneck in high-performance distributed systems, so building a fast sequencer is an important step to improving whole-system performance.

Our sequence server runs on a single machine and provides an increasing 8-byte integer to client processes running on remote machines. The baseline design uses HERD RPCs. The worker threads at the server share an 8-byte counter; each client can send a sequencer request to any worker. The worker's application processing consists of atomically incrementing the shared counter by one. When Doorbell batching is enabled, we use an additional application-level optimization to reduce contention for the shared counter: after collecting N requests, a worker atomically increments the shared counter by N, thereby claiming ownership of a sequence of N consecutive integers. It then sends these N integers to the clients using a batched Doorbell (one integer per client).

	Baseline	+RPC opts	`Spec-S0`	Atomics
Throughput	26	97.2	122	2.24
Bottleneck	CPU	DMA bw	NIC	PCIe RTT

Table 3: Sequencer throughput (Mrps) and bottlenecks on CIB

Figure 7 shows the effect of batching and multi-queue on the HERD RPC-based sequencer's throughput with an increasing number of server CPU cores. We run 1 worker thread per core and use 70 client processes on 5 client machines. Batching increases single-core throughput from 7.0 million requests per second (Mrps) to 16.6 Mrps. In this mode, each core still uses 2 response UD queue pairs—one for each NIC port—and is bottlenecked by the NIC processing units handling the QPs; engaging more PUs with multi-queue (3 per-port QPs per core) increases core throughput to 27.4 Mrps. With 6 cores and both optimizations, throughput increases to 97.2 Mrps and is bottlenecked by DMA bandwidth: The DMA bandwidth limit for the batched UD SENDs used by our sequencer is 101.6 million operations/s (Section 5.2.1). At 97.2 Mrps, the sequencer is within 5% of this limit; we attribute the gap to PCIe link- and physical-layer overheads in the DMA-ed requests, which are absent in our SEND-only benchmark. When more than 6 cores are used, throughput drops because the response batch size are smaller: With 6 cores (97.2 Mrps), there are 15.9 responses per batch; with 10 cores (84 Mrps), there are 4.4 responses per batch.

4.2.1 Sequencer-specific optimizations

The above design is a straightforward adoption of general-purpose RPCs for a sequencer, and inherits the limitations of the RPC protocol. First, the connected QPs used for writing requests require state in the server's NIC and limit scalability to a few hundred RPC clients [20]. Higher scalability necessitates exclusive use of datagram transport which only supports SEND/RECV verbs. The challenge then is to use SEND/RECV instead of WRITEs for sequencer *requests* without sacrificing server performance. Second, it uses PCIe inefficiently: UD SEND work queue elements on Mellanox's NICs span $\geq$ 2 cache lines because of their 68-byte header (Table 1); sending 8 bytes of useful sequencer data requires 128 bytes (2 cache lines) to be DMA-ed by the NIC.

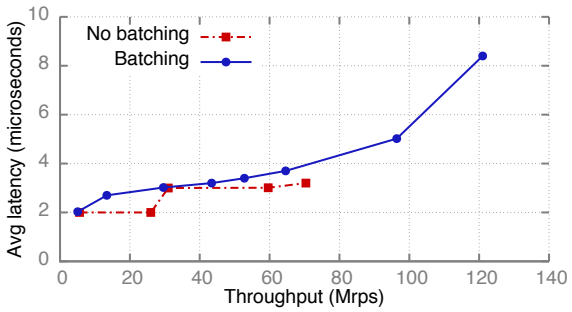


Figure 8: Impact of response batching on `Spec-S0` latency

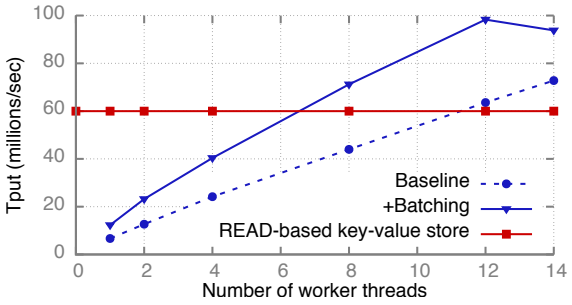


Figure 9: Improvement in HERD's throughput with 5% PUTs

We exploit the specific requirements of the sequencer to overcome these limitations. We use header-only SENDs for both requests and responses to solve both problems:

1. The client's header-only request SENDs generate header-only, single-DMA RECVs at the server (Figure 6), which are as fast as the WRITEs used earlier.
2. The server's header-only response SEND WQEs use a header field for application payload and fit in 1 cache line (Figure 5), reducing the data DMA-ed per response by 50% to 64 bytes.

Using header-only SENDs requires encoding application information in the SEND packet header; we use the 4-byte *immediate integer* field of RDMA packets [1]. Our 8-byte sequencer works around the 4-byte limit as follows: Clients speculate the 4 higher bytes of the counter and send it in a header-only SEND. If the client's guess is correct, the server sends the 4 lower bytes in a header-only SEND, else it sends the entire 8-byte value in a regular, non header-only SEND which later triggers an update of the client's guess. Only a tiny fraction ($\leq C/2^{32}$ with C clients) of SENDs are regular. We discuss this speculation technique further in Section 5.

We call this datagram-only sequencer `Spec-S0` (speculation with header-only SENDs). Figure 7 shows its throughput with increasing server CPU cores. `Spec-S0`'s DMA bandwidth limit is higher than HERD RPCs because of smaller response WQEs; it achieves 122 Mrps and is limited by the NIC's processing power instead of PCIe bandwidth. `Spec-S0` has lower single-core throughput than the HERD RPC-based sequencer because of the additional CPU overhead of posting RECVs.

4.2.2 Latency

Figure 8 shows the average end-to-end latency of `Spec-S0` with and without response batching. Both modes receive a batch of requests from the NIC; the two modes differ only in the method used to send responses. The non-batched mode sends responses one-by-one using WQE-by-MMIO whereas the batched mode uses Doorbell when multiple responses are available to send. We batch atomic increments to the shared counter in both modes. We use 10 server CPU cores, which is the minimum required to achieve peak throughput. We measure throughput with increasing client load by adding more clients, and by increasing the number of outstanding requests per client. Batching adds up to 1 µs of latency because of the additional DMA read required with the Doorbell method. We believe that the small additional latency is acceptable because of the large throughput and CPU efficiency gains from batching.

4.2.3 Atomics-based sequencers

Atomic fetch-and-add over RDMA is an appealing method to implement a sequencer: Binnig et al. [11] use this design for the timestamp server in their distributed transaction protocol. However, lock contention for the counter among the NIC's PUs results in poor performance. The effects of contention are exacerbated by the *duration* for which locks are held—several hundred nanoseconds for PCIe round trips. Our RPC-based sequencers have lower contention and shorter lock duration: the programmability of general-purpose CPUs allows us to batch updates to the counter which reduces cache line contention, and proximity to the counter's storage (i.e., core caches) makes these updates fast. Figure 7 shows the throughput of our atomics-based sequencer: it achieves only 2.24 Mrps, which is 50x worse than our optimized design, and 12.2x worse than our single-core throughput. Table 3 summarizes the performance of our sequencers.

4.3 Key-value stores

Several designs have been proposed for RDMA-based key-value storage. The HERD key-value cache uses HERD RPCs for all requests and does not bypass the remote CPU; other key-value designs bypass the remote CPU for key-value GETs (Pilaf [24] and FaRM-KV [13]), or for both GETs and PUTs (DrTM-KV [30] and Nessie [27]). Our goal here is to demonstrate how our guidelines can be used to optimize or find flaws in RDMA system designs in general; we do not compare across different key-value systems. We first present performance improvements for HERD. Then, we show how the performance of atomics-based key-value stores is adversely affected by lock contention inside NICs.

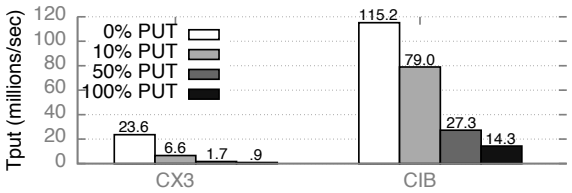


Figure 10: Throughput of emulated DrTM-KV with increasing updates.

4.3.1 Improving HERD's performance

We apply batching to HERD as follows. After collecting $N \leq C$ requests, the server-side worker performs the GET or PUT operations on the backing datastore. It uses prefetching to hide the memory latency of accesses to the storage data structures [32, 23]. Then, the worker sends the responses either one-by-one using WQE-by-MMIO, or as a single batch using Doorbell.

For evaluation, we run a HERD server with a variable number of workers on a CIB machine; we use 128 client threads running on eight client machines to issue requests. We pre-populate the key space partition owned by each worker with 8 million key-value pairs, which map 16-byte keys to 32-byte values. The workload consists of 95% GET and 5% PUT operations, with keys chosen uniformly at random from the inserted keys.

Figure 9 shows the throughput achieved in the above experiment. We also include the maximum throughput achievable by a READ-based key-value store such as Pilaf or FaRM-KV that uses ≥ 2 small READs per GET (one READ for fetching the index entry, and one for fetching the value). We compute this analytically by halving CIB's peak inbound READ throughput (Section 5.3.1). We make three observations:

- Batching improves HERD's per-core throughput by 83% from 6.7 Mrps to 12.3 Mrps. This improvement is smaller than for the sequencer because the CPU processing time saved from avoiding MMIOs is smaller relative to per-request processing in HERD than in the sequencer.
- Batching improves peak throughput by 35% from 72.8 Mrps to 98.3 Mrps. Batched throughput is bottlenecked by PCIe DMA bandwidth.
- With batching, HERD's throughput is up to 63% higher than a READ-based key-value store. While HERD's original non-batched design requires 12 cores to outperform a READ-based design, only 7 cores are needed with batching. This highlights the importance of including low-level factors such as batching when comparing RDMA system designs.

4.3.2 Atomics-based key-value stores

DrTM-KV [30] and Nessie [27, 28] use RDMA atomics to bypass the remote CPU for both GETs and PUTs. However, these projects do not consider the impact of the NIC's concurrency control on performance, and present performance for either GET-only (DrTM-KV) or GET-mostly workloads (Nessie). We now show that locking inside the NIC results in low PUT throughput, and degrades throughput even when only a small fraction of key-value operations are PUTs.

We discuss DrTM-KV here because of its simplicity, but similar observations apply to Nessie. DrTM-KV caches some fields of its key-value index at all clients; GETs for cached keys use one READ. PUT operations lock, update, and unlock key-value items; locking and unlocking is done using atomics. Running DrTM-KV's codebase on CIB requires significant modification because CIB's dual-port NIC are connected in a way that does not allow cross-port communication. To overcome this, we wrote a simplified emulated version of DrTM-KV: we emulate GETs with 1 READ and PUTs with 2 atomics, and assume a 100% cache hit rate.

Figure 10 shows the throughput of our emulated DrTM-KV server with different fractions of PUT operations in the workload. The server hosts 16 million items with 16-byte keys and 32-byte values. Clients use randomly chosen keys and we use as many clients as required to maximize throughput. Although throughput for a 100% GET workload is high, adding only 10% PUTs degrades it by 72% on CX3 and 31% on CIB. Throughput with 100% PUTs is a tiny fraction of GET-only throughput: 4% on CX3 and 12% on CIB. Note that the degradation for CIB is more gradual than for CX3 because CIB has a better locking mechanism, as shown in Section 5.

5 Low-level factors in RDMA

Our guidelines and system designs are based on an improved understanding of low-level factors that affect RDMA performance, including I/O initiation mechanisms, PCIe, and NIC architecture. These factors are complicated and there is little existing literature describing them or studying their relevance to networked systems. We attempt to fill this void by presenting clarifying performance measurements, experiments, and models; Table 4 shows a partial summary for CIB. Additionally, we discuss the importance of these factors to general RDMA system design beyond the two systems in Section 4.

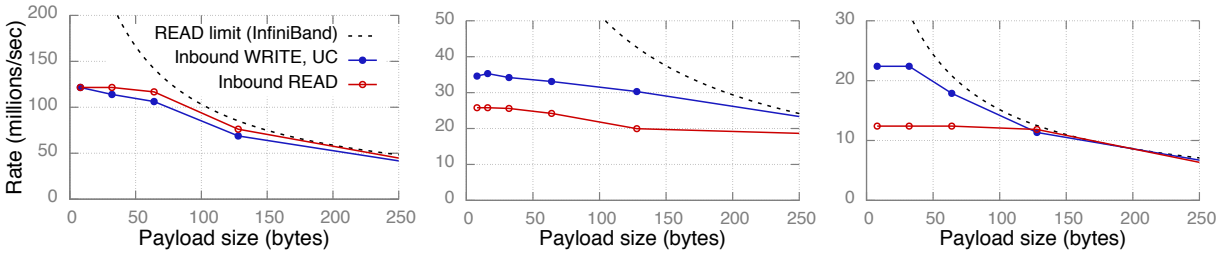
We divide our discussion into three common use cases that highlight different low-level factors: (1) batched operations, (2) non-batched operations, and (3) atomic operations. For each case, we present a performance analysis focusing on hardware bottlenecks and optimizations, and discuss implications on RDMA system design.

5.1 PCIe models

We have demonstrated that understanding the PCIe behavior is critical for improving RDMA performance. How-

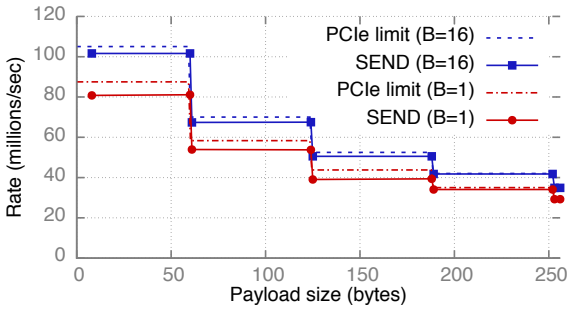
	Outbound verbs			Inbound verbs					
	UD SENDs			UD RECVs		READs		Atomics	
	Non-batch	Batch	Batch + HO	Batch	Batch + HO	≤ 64 B	128 B	$Z = 1$	$Z \geq 4096$
Rate (Mops)	80	101.6	157	82	122	121.6	76.2	2.24	52
Bottleneck	MMIO bw	DMA bw	NIC	NIC	NIC	NIC	IB bw	PCIe RTT	NIC

Table 4: Throughput and bottleneck of different modes of RDMA verbs on CIB. HO denotes the header-only optimization.

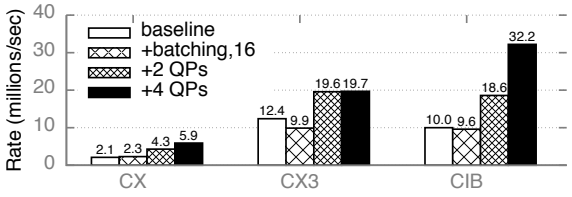


(a) CIB *(b) CX3* *(c) CX*

Figure 12: Inbound READ and UC WRITE throughput, and the *InfiniBand* limit for READs. Note the different scales for Y axes.



(a) PCIe limits for UD SEND with batch size B



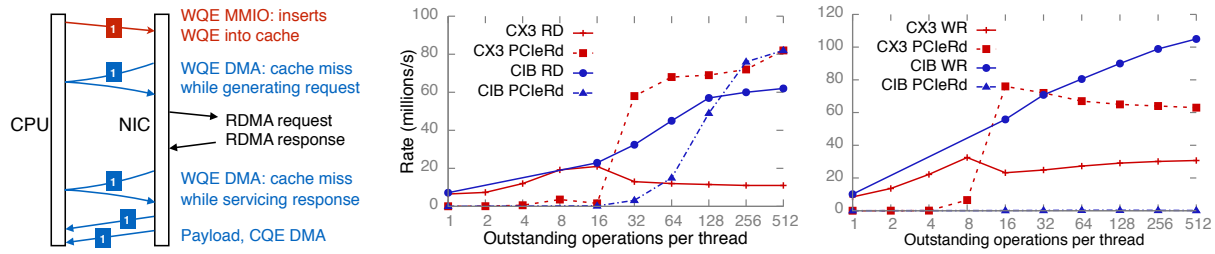
(b) Optimizations for UD SEND

Figure 11: *(a)* Peak batched and non-batched UD SEND throughput on CIB, with batch size B; dotted lines show corresponding PCIe limits. *(b)* Effect of optimizations on single-core UD SEND throughput with 60-byte payload (128-byte WQE).

ever, deriving analytical models of PCIe behavior without access to proprietary/confidential NIC manuals and our limited resources—per–cache line PCIe counters and undocumented driver software—required extensive experimentation and analysis. Our derived models are presented in a slightly simplified form at several points in this paper (Figures 5, 6, 13). The exact analytical models are complicated and depend on several factors such as the NIC, its PCIe capability, the verb and transport, the level of Doorbell batching, etc. To make our models easily accessible, we instrumented the datapath of two Mellanox drivers (ConnectX-3 and Connect-IB) to provide statistics about PCIe bandwidth use (https://github.com/efficient/rdma_bench). Our models and drivers are restricted to requester-side PCIe behavior. We omit responder-side PCIe behavior because it is the same as described in our previous work [20]: inbound READs and WRITEs generate one PCIe read and write, respectively; inbound SENDs trigger a RECV completion—we discuss the PCIe transactions for the RECV.

5.2 Batched operations

A limitation of batching on current hardware makes it useful mainly for datagram transport: all operations in a batch must use the same queue pair because Doorbells are per QP. This limitation seems fundamental to the parallel architecture of NICs: In a hypothetical NIC design where Doorbells contained information relevant for multiple queue pairs (e.g., a compact encoding of "2 and 1 new WQEs for QP 1 and QP 2, respectively"), sending the Doorbell to the NIC processing units handling these QPs would require an expensive selective broadcast inside the NIC. These PUs would then issue separate DMAs for WQEs, losing the coalescing advantage of batching. This limitation makes batching less useful for connected QPs, which provide only one-to-one communication between two machines: the chances that a process has multiple messages for the same remote machine are low in large deployments. We therefore discuss batching for UD transport only.



(a) PCIe model for reliable verbs *(b) WQE cache misses for READs* *(c) WQE cache misses for RC WRITEs*

Figure 13: PCIe model showing possible WQE cache misses, and measurement of WQE cache misses for READs and RC WRITEs.

5.2.1 UD SENDs

Figure 11 shows the throughput and PCIe bandwidth limit of batched and non-batched UD SENDs on CIB. We use one server to issue SENDs to multiple client machines. With batching, we use batches of size 16 (i.e., the NIC DMAs 16 work queue elements per Doorbell). Otherwise, the CPU writes WQEs by the WQE-by-MMIO method. We use as many cores as required to maximize throughput. Batching improves peak SEND throughput by 27% from 80 million operations/s (Mops) to 101.6 Mops.

Bottlenecks Batched throughput is limited by DMA bandwidth. For every DMA completion of size C_{rc} bytes, there is header overhead of P_c bytes (Section 2.1), leading to 13443 MB/s of useful DMA read bandwidth on CIB. As UD WQEs span at least 2 cache lines, the maximum WQE transfer rate is 13443/128 = 105 million/s, which is within 5% of our achieved throughput; we attribute the difference to link- and physical-layer PCIe overheads.

Non-batched throughput is limited by MMIO bandwidth. The write-combining MMIO rate on CIB is $(16 * P_{bw})/(64 + P_r) = 175$ million cache lines/s. UD SEND WQEs with non-zero payload span at least 2 cache lines (Table 1), and achieve up to 80 Mops. This is within 10% of the 87.5 Mops bandwidth limit.

Multi-queue optimization Figure 11b shows single-core throughput for batched and non-batched 60-byte UD SENDs—the largest payload size for which the WQEs fit in 2 cache lines. Interestingly, batching *decreases* core throughput if only one QP is used: with one QP, a core is coupled to a NIC processing unit (Section 3.3), so throughput depends on how the PU handles batched and non-batched operations. Batching has the expected effect when we break this coupling by using multiple QPs. Batched throughput increases by ~ 2x on all clusters with 2 QPs, and between 2–3.2x with 4 QPs. Non-batched (WQE-by-MMIO) throughput *does not* increase with multiple QPs (not shown in graph), showing that it is CPU-limited.

	RECV 0	RECV ≥ 1	SEND 0	SEND ≥ 1
CIB	122.0	82.0	157.0	101.6
CX3	34.0	21.8	32.1	26.0
CX	15.3	9.6	11.9	11.9

Table 5: Per-NIC rate (millions/s) for header-only (0) and regular (≥ 1) SENDs and RECVs

Design implications RDMA-based systems can often choose between CPU-bypassing and CPU-involving designs. For example, clients can access a key-value store either by READing directly from the server's memory [24, 13, 30, 28], or via RPCs as in HERD [20]. Our results show that achieving peak performance on even the most powerful NICs *does not* require a prohibitive amount of CPU power: only 4 cores are needed to saturate the fastest PCIe links. Therefore, CPU-involving designs will not be limited by CPU processing power, provided that their application-level processing permits so.

5.2.2 UD RECVs

Table 5 compares the throughput of header-only and payload-carrying regular RECVs (Figure 6). In our experiment, multiple client machines issue SENDs to one server machine that posts RECVs. On CIB, avoiding the payload DMA with header-only RECVs increases throughput by 49% from 82 Mops to 122 Mops, and makes them as fast as inbound WRITEs (Figure 12a).[4] Table 5 also compares header-only and regular SENDs (Figure 5). Header-only SENDs use single-cache line WQEs and achieve 54% higher throughput.

Design implications Developers avoid RECVs at performance critical machines as a rule of thumb, favoring the faster READ/WRITE verbs [24, 20]. Our work provides the exact reason: RECVs are slow due to the CQE DMA; they are as fast as inbound WRITEs if it is avoided.

Speculation Current RDMA implementations allow 4 bytes of application data in the packet header of header-

[4]Inline-receive improves regular RECV throughput from 22 Mops to 26 Mops on CX3, but is not yet supported for UD on CIB.

only SENDs. For applications that require larger messages, header-only SEND/RECV can be used if speculation is possible; we demonstrated such a design for an 8-byte sequencer in Section 4.2. In general, speculation works as follows: clients transmit their expected response along with requests, and get a small confirmation response in the common case. For example, in a key-value store with client-side caching, clients can send GET requests with the key and its cached version number (using a WRITE or regular SEND). The server replies with a header-only "OK" SEND if the version is valid.

There are applications for which 4 bytes of per-message data suffices. For example, some database tables in the TPC-C [29] benchmark have primary key size between 2 and 3 bytes. A table access request can be sent using a header-only SEND (using the remaining 1 byte to specify the table ID), while the response may need a larger SEND.

5.3 Non-batched operations

5.3.1 Inbound READs and WRITEs

Figure 12 shows the measured throughput of inbound READs and UC WRITEs, and the InfiniBand bandwidth limit of inbound READs. We do not show the InfiniBand limit for WRITEs and the PCIe limits as they are higher.

Bottlenecks On our clusters, inbound READs and WRITEs are initially bottlenecked by the NIC's processing power, and then by InfiniBand bandwidth. The payload size at which bandwidth becomes a bottleneck depends on the NIC's processing power relative to bandwidth. For READs, the transition point is approximately 128 bytes, 256 bytes, and 64 bytes for CX, CX3, and CIB, respectively. CIB NICs are powerful enough to saturate 112 Gbps with 64-byte READs, whereas CX3 NICs require 256-byte READs to saturate 56 Gbps.

Implications The transition point is an important factor for systems that make tradeoffs between the size and number of READs: For key-value lookups of small items (~32 bytes), FaRM's key-value store [13] can use one large (~256-byte) READ. In a client-server design where inbound READs determine GET performance, this design performs well on CX3 because 32- and 256-byte READs have similar throughput; other designs such as DrTM-KV [30] and Pilaf [24] that instead use 2–3 small READs may provide higher throughput on CIB.

5.3.2 Outbound READs and WRITEs

For brevity, we only present a summary of the performance of non-batched outbound operations on CIB. Outbound UC WRITEs larger than 28 bytes, i.e., WRITEs with WQEs spanning more than one cache line (Table 1), achieve up to 80 Mops and are bottlenecked by PCIe MMIO throughput, similar to non-batched outbound SENDs (Figure 11a). READs achieve up to 88 Mops and

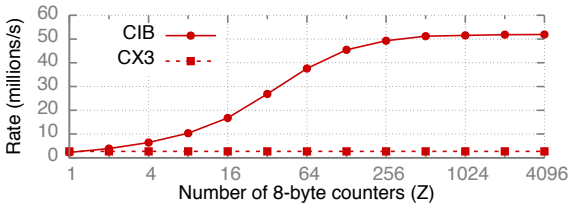


Figure 14: Atomics throughput with increasing concurrency

are bottlenecked by NIC processing power.

Achieving high outbound throughput requires maintaining multiple outstanding requests via pipelining. When the CPU initiates an RDMA operation, the work queue element is inserted into the NIC's WQE cache. However, if the CPU injects new WQEs faster than the NIC's processing speed, this WQE can be evicted by newer WQEs. This can cause cache misses when the NIC eventually processes this WQE while generating its RDMA request packets, while servicing its RDMA response, or both. Figure 13a summarizes this model.

To quantify this effect, we conduct the following experiment on CIB: 14 requester threads on a server issue windows of N 8-byte READs or WRITEs over reliable transport to 14 remote processes. In Figures 13b and 13c, we show the cumulative RDMA request rate, and the extent of WQE cache misses using the `PCIeRdCur` counter rate. Each thread waits for the N requests to complete before issuing the next window. We use all 14 cores on the server to generate the maximum possible request rate, and RC transport to include cache misses generated while processing ACKs for WRITEs. We make the following observations, showing the importance of the WQE cache in improving and understanding RDMA throughput:

- The optimal window size for maximum throughput is not obvious: throughput does not always increase with increasing window size, and is dependent on the NIC. For example, $N = 16$ and $N = 512$ maximize READ throughput on CX3 and CIB respectively.
- Higher RDMA throughput may be obtained at the cost of PCIe reads. For example, on CIB, both READ throughput and PCIe read rate increases as N increases. Although the largest N is optimal for a machine that only issues outbound READs, it may be suboptimal if it also serves other operations.
- CIB's NIC can handle the CPU's peak WQE injection rate for WRITEs and never suffers cache misses. This is not true for READs, indicating that they require more NIC processing than reliable WRITEs.

5.4 Atomic operations

NIC processing units contend for locks during atomic operations (Section 3.4). The performance of atomics depends on the amount of parallelism in the workload with respect to the NIC's internal locking scheme. To vary

the amount of parallelism, we create an array of Z 8-byte counters in a server's memory, and multiple remote client processes issue atomic operations on counters chosen randomly at each iteration. Figure 14 shows the total client throughput in this experiment. For CX3, it remains 2.7 Mops irrespective of Z; for CIB, it rises to 52 Mops.

Inferring the locking mechanism The flatness of CX3's throughput graph indicates that it serializes all atomic operations. For CIB, we measured performance with randomly chosen pairs of addresses and observed lower performance for pairs where both addresses have the same 12 LSBs. This strongly suggests that CIB uses 4096 buckets to slot atomic operations by address—a new operation waits until its slot is empty.

Bottlenecks and implications Throughput on CX3 is limited by PCIe latency because of serialization. For CIB, buffering and computation needed for PCIe read-modify-write makes NIC processing power the bottleneck.

The abysmal throughput for $Z = 1$ on both NICs reaffirms that atomics are a poor choice for a sequencer; our optimized sequencer in Section 4 provides 12.2x higher performance with a *single* server CPU core. A lock service for data stores, however, might use a larger Z. Atomics could perform well if such an application used CIB, but they are very slow with CX3, which is the NIC used in prior work [27, 30]. With CIB, careful lock placement is still necessary. For example, if page-aligned data records have their lock variables at the same offset in the record, all lock requests will have the same 12 LSBs and will get serialized. A deterministic scheme that places the lock at different offsets in different records, or a scheme that keeps locks separate from the data will perform better.

6 Related work

High-performance RDMA systems Designing high-performance RDMA systems is an active area of research. Recent advances include several key-value storage systems [24, 13, 20, 30, 28] and distributed transaction processing systems [30, 12, 14, 11]. A key design decision in each of these systems is the choice of verbs, made using a microbenchmark-based performance comparison. Our work shows that there are more dimensions to these comparisons than these projects explore: two verbs cannot be exhaustively compared without exploring the space of low-level factors and optimizations, each of which can offset verb performance by several factors.

Low-level factors in network I/O Although there is a large body of work that measures the throughput and CPU utilization of network communication [18, 16, 26, 13, 20], there is less existing literature on understanding the low-level behavior of network cards. NIQ [15] presents a high-level picture of the PCIe interactions between an Ethernet NIC and CPUs, but does not discuss the more subtle interactions that occur during batched transfers. Lee et al. [22] study the PCIe behavior of Ethernet cards using a PCIe protocol analyzer, and divide the PCIe traffic into Doorbell traffic, Ethernet descriptor traffic, and actual data traffic. Similarly, analyzing RDMA NICs using a PCIe analyzer may reveal more insights into their behavior than what is achievable using PCIe counters.

7 Conclusion

Designing high-performance RDMA systems requires a deep understanding of low-level RDMA details such as PCIe behavior and NIC architecture: our best sequencer is ~50x faster than an existing design and scales perfectly, our optimized HERD key-value store is up to 83% faster than the original, and our fastest transmission method is up to 3.2x faster than the commonly-used baseline. We believe that by presenting clear guidelines, significant optimizations based on these guidelines, and tools and experiments for low-level measurements on their hardware, our work will encourage researchers and developers to develop a better understanding of RDMA hardware before using it in high-performance systems.

Acknowledgments We are tremendously grateful to Joseph Moore and NetApp for providing access to the CIB cluster. We thank Hyeontaek Lim and Sol Boucher for providing feedback, and Liuba Shrira for shepherding. Emulab [31] and PRObE [17] resources were used in our experiments. PRObE is supported in part by NSF awards CNS-1042537 and CNS-1042543 (PRObE). This work was supported by funding from the National Science Foundation under awards 1345305 and 1314721, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC).

Appendix A. WQE-by-MMIO and Doorbell PCIe use

We denote the doorbell size by d. The total data transmitted from CPU to NIC with the WQE-by-MMIO method is $T_{bf} = 10 * (\lceil 65/64 \rceil * (64 + P_r))$ bytes. With cache line padding, 65-byte WQEs are laid out in 128-byte slots in host memory; assuming $C_{rc} = 128$, $T_{db} = (d+P_r)+(10*(128+P_c))$ bytes. We ignore the PCIe link-layer traffic since it is small compared to transaction-layer traffic: it is common to assume 2 link-layer packets (1 flow control update and 1 acknowledgment, both 8 bytes) per 4-5 TLPs [9], making the link-layer overhead < 5%. Substituting $d = 8$ gives $T_{bf} = 1800$, and $T_{db} = 1534$.

References

[1] Infiniband architecture specification volume 1. https://cw.infinibandta.org/document/dl/7859.

[2] Intel Atom Processor C2000 Product Family for Microserver. http://www.intel.in/content/dam/www/public/us/en/documents/datasheets/atom-c2000-microserver-datasheet.pdf.

[3] Intel Xeon Processor E5-1600/2400/2600/4600 v3 Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v3-datasheet-vol-2.pdf.

[4] Intel Xeon Processor E5-1600/2400/2600/4600 (E5-Product Family) Product Families. http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-1600-2600-vol-2-datasheet.pdf.

[5] Intel Xeon Processor D-1500 Product Family. http://www.intel.in/content/dam/www/public/us/en/documents/product-briefs/xeon-processor-d-brief.pdf.

[6] Intel Xeon Phi Processor Knights Landing Architectural Overview. https://www.nersc.gov/assets/Uploads/KNL-ISC-2015-Workshop-Keynote.pdf.

[7] Mellanox ConnectX-4 product brief. http://www.mellanox.com/related-docs/prod_silicon/PB_ConnectX-4_VPI_Card.pdf.

[8] Mellanox OFED for linux user manual. http://www.mellanox.com/related-docs/prod_software/Mellanox_OFED_Linux_User_Manual_v2.2-1.0.1.pdf.

[9] Understanding Performance of PCI Express Systems. http://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

[10] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: a shared log design for flash clusters. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[11] C. Binnig, U. Çetintemel, A. Crotty, A. Galakatos, T. Kraska, E. Zamanian, and S. B. Zdonik. The end of slow networks: It's time for a redesign. *CoRR*, abs/1504.01048, 2015. URL http://arxiv.org/abs/1504.01048.

[12] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proc. 11th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2016.

[13] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[14] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[15] M. Flajslik and M. Rosenblum. Network interface design for low latency request-response protocols. In *Proc. USENIX Annual Technical Conference*, June 2013.

[16] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. Comparison of frameworks for high-performance packet io. In *ANCS*, 2015.

[17] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. PRObE: A Thousand-Node Experimental Cluster for Computer Systems Research.

[18] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, Aug. 2010.

[19] S. Hauger, T. Wild, A. Mutter, A. Kirstaedter, K. Karras, R. Ohlendorf, F. Feller, and J. Scharf. Packet processing at 100 Gbps and beyond - challenges and perspectives. In *Photonic Networks, 2009 ITG Symposium on*, 2009.

[20] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. ACM SIGCOMM*, Aug. 2014.

[21] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using GPUs in software packet processing. In *Proc. 12th USENIX NSDI*, May 2015.

[22] S. Larsen and B. Lee. Platform io dma transaction acceleration. In *CACHES*. ACM, 2011.

[23] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Apr. 2014.

[24] C. Mitchell, Y. Geng, and J. Li. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *Proc. USENIX Annual Technical Conference*, June 2013.

[25] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Apr. 2013.

[26] L. Rizzo. netmap: a novel framework for fast packet I/O. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, June 2012.

[27] T. Szepesi, B. Wong, B. Cassell, , and T. Brecht.

Designing a low-latency cuckoo hash table for write-intensive workloads. In *WSRC*, 2014.

[28] T. Szepesi, B. Cassell, B. Wong, T. Brecht, and X. Liu. Nessie: A decoupled, client-driven, key-value store using RDMA. Technical Report CS-2015-09, University of Waterloo, David R. Cheriton School of Computer Science, Waterloo, Canada, June 2015.

[29] TPC-C. TPC benchmark C. http://www.tpc.org/tpcc/.

[30] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[31] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Dec. 2002.

[32] D. Zhou, B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. 9th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Dec. 2013.

Balancing CPU and Network in the Cell Distributed B-Tree Store

Christopher Mitchell Kate Montgomery Lamont Nelson Siddhartha Sen* Jinyang Li

New York University **Microsoft Research*

{cmitchell, kem493, lamont.nelson, jinyang}@cs.nyu.edu, sidsen@microsoft.com

Abstract

In traditional client-server designs, all requests are processed at the server storing the state, thereby maintaining strict locality between computation and state. The adoption of RDMA (Remote Direct Memory Access) makes it practical to relax locality by letting clients fetch server state and process requests themselves. Such client-side processing improves performance when the server CPU, instead of the network, is the bottleneck. We observe that combining server-side and client-side processing allows systems to balance and adapt to the available CPU and network resources with minimal configuration, and can free resources for other CPU-intensive work.

We present Cell, a distributed B-tree store that combines client-side and server-side processing. Cell distributes a global B-tree of "fat" (64MB) nodes across machines for server-side searches. Within each fat node, Cell organizes keys as a local B-tree of RDMA-friendly small nodes for client-side searches. Cell clients dynamically select whether to use client-side or server-side processing in response to available resources and the current workload. Our evaluation on a large RDMA-capable cluster show that Cell scales well and that its dynamic selector effectively responds to resource availability and workload properties.

1 Introduction

In the traditional client-server design, the server does the vast majority of the computation, storing state and processing operations by performing computation over the state. Clients simply send RPC requests and receive replies from the server. Maintaining strict locality of computation and state is crucial for performance when the cost of communication is high.

Commodity clusters have recently started to embrace ultra-low-latency networks with Remote Direct Memory Access (RDMA) support [3, 1, 7], just as in-memory storage has become practical and performant [34, 37, 31, 45]. RDMA is supported over InfiniBand or over Ethernet via RoCE [46, 49], each of which offers high throughput (20-100 Gbps) and low round-trip latency (a few microseconds). With RDMA, a machine can directly read or write parts of a peer's memory without involving the remote machine's CPU or the local kernel; traditional message passing can also still be used.

As a result, RDMA has drastically lowered the cost of communication, thereby permitting an alternative system design that relaxes locality between computation and state. Clients can process requests by fetching server state using RDMA and performing computation on the state themselves [7, 32]. Client-side processing consumes similar total CPU resources to server-side processing, except with the CPU load shifted to the clients. However, this flexibility comes at a cost: fetching server state consumes extra network resources, which may become a bottleneck. For datacenters with capable networks, the bottleneck network resource is each server's NIC(s).

Several existing in-memory distributed storage systems utilize RDMA-capable networks; all are unsorted key-value stores that exclusively use client-side or server-side processing. For example, Pilaf [32] and FaRM [7] rely on client-side processing for all read operations. HERD [22] uses only server-side processing. None of these solutions is satisfactory: we recognize that practical in-memory storage systems exist on a continuum between being CPU-bound and network-bound that also shifts as workloads change. Therefore, we explore a hybrid approach that augments server-side processing with client-side operations whenever bypassing the server CPU leads to better performance.

Modern bare-metal servers are usually equipped with as many CPU cores as necessary to saturate the server's NIC [2]. Nevertheless, there are several common scenarios where servers' CPUs can become bottlenecked, causing client-side processing to be more desirable. First, when deploying a distributed storage system in the cloud, the virtual machines' CPUs must be explicitly rented. As a result, one usually reserves a few CPU cores that are sufficient for the average load in order to save costs. Doing so leaves servers overloaded during load spikes[1]. Second, a networked system may be deployed in a shared cluster where the same physical machines running the servers also run other CPU-intensive jobs, including storage-related application logic, to maximize cluster utilization. In this case, one also does not want to assign

[1] One can dynamically add more server instances on-the-fly in case of server overload. However, this would be too slow to handle load spikes of a few seconds.

more CPU cores than necessary for the servers to handle the average load, thereby also resulting in server CPU overload during load spikes.

In this paper, we investigate how to balance CPU and network by building a distributed, in-memory B-tree store, called Cell. We choose a B-tree as a case study system because of its challenges and importance: as a sorted data structure, B-trees serve as the storage backend for distributed databases. A distributed B-tree spreads its nodes across many servers, and supports get, put, delete, and range operations. The key component underlying all operations is search, i.e. traversing the B-tree: Cell combines client-side and server-side processing for B-tree searches. Cell builds a hierarchical B-tree: Cell distributes a B-tree of "fat" nodes (*meganodes*), each containing a local B-tree of small nodes, across server machines. This hierarchical design enables simultaneous server-side search through meganodes and efficient client-side search through small nodes. Our architecture permits reliable lock-free search with caching even in the face of *concurrent* writes and structural changes to the tree. Our Cell prototype also provides distributed transactions on top of the B-tree using traditional techniques; we omit a discussion of this feature due to scope.

In order to arrive at the best balance between using client-side and server-side processing, Cell needs to dynamically adjust its decision. Ideally, when server CPUs are not saturated, Cell clients should choose server-side searches. When a server becomes overloaded, some but not all clients should switch to performing client-side searches. The goal is to maximize the overall search throughput. We model system performance using basic queuing theory; each client independently estimates the "bottleneck queuing delays" corresponding to both search types, then selects between server-side and client-side search accordingly. This dynamic selection strategy achieves the best overall throughput across a spectrum of different ratios of available server CPU and network resources.

We have implemented a prototype of Cell running on top of RDMA-capable InfiniBand. Experiments on the PRObE Nome cluster show that Cell scales well across machines. With 16 server machines each consuming 2 CPU cores, Cell achieves 5.31 million ops/sec combining both server-side and client-side searches, 65% faster than server-side search alone while still leaving the remaining 6 cores per machine for CPU-intensive application logic. More importantly, Cell balances servers' CPU and network resources, and is able to do so in different environments. Cell clients make good dynamic decisions on when to use client-side searches, consistently matching or exceeding the best manually-tuned fixed percentages of client-side and server-side searches or either of the search types alone. The system responds quickly (in $< 1s$) and correctly to maintain low operation latency in the face of load spikes and tree structure modifications.

We present Cell's design in Section 2 and describe implementation-specific details in Section 3. We thoroughly evaluate Cell's performance and design choices in Section 4. We explore related systems in Section 5.

2 Cell Design

Cell provides a sorted in-memory key-value store in the form of a distributed B-tree. In this section, we give an overview of Cell (2.1) and then discuss the main components of our design: our B-tree structure (2.2) and our hybrid search technique (2.3, 2.5).

2.1 Overview

In designing the Cell distributed B-tree store, we make two high-level design decisions: (1) support *both* client-side and server-side operations; (2) restrict client-side processing to read-only operations, including B-tree searches and key-value fetches. (1) is made possible by a hierarchical B-tree of B-trees. (2) is logical because practical workloads are search- and read-heavy [34], and client-side writes would involve much more complexity. As Section 4.5 shows, we can reap the benefits of client-side processing across a variety of workloads with different fractions of read vs. write operations. Cell's basic design faces two novel challenges: how to ensure the correctness of RDMA searches during concurrent server-side modification, and when should clients prefer client-side over the default server-side processing? Although a large body of existing work explores concurrent B-trees, these works assume that servers' CPUs still have full control over access to the servers' data. With RDMA, the storage system must provide its own techniques to synchronize reads and writes, or at least ensure that reads are performed over consistent data.

Cell organizes data in a hierarchical B-tree of B-trees to ensure that both server-side and RDMA-based searches are efficient. At the cluster level, Cell builds a B-tree out of fat nodes called *meganodes*, containing tens or hundreds of megabytes of structural metadata. Meganodes are spread across the memory of the servers in the cluster. Within each meganode, Cell builds a local B-tree consisting of small nodes (e.g. 1KB). The local B-tree allows a server to search efficiently for a key within a meganode, while simultaneously allowing remote clients to search within a meganode using a small number of efficient RDMAs. Other systems like BigTable [5] only support server-side search, so they can use different searchable local data structures such as skiplists that would require many more roundtrips to access remotely.

We adopt the common practice of storing the data of a B-tree at its leaf level. Thus, the leaf meganodes of the tree store local pointers to data while the internal megan-

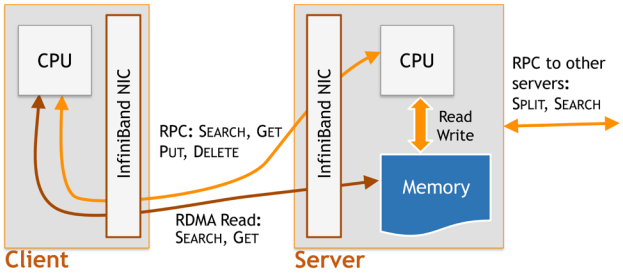


Figure 1: Cell's architecture and interactions (Section 2).

odes store remote pointers to other meganodes on remote machines. All pointers consist of a region ID and offset. Each server's in-memory state includes multiple large contiguous memory areas (e.g. 1GB) containing node regions that hold meganodes or extents regions holding key-value data. All clients and servers maintain a cache of the global region table that maps region IDs to the IP addresses of the responsible machines. Servers also actively exchange region information with each other asynchronously. We assume that server membership is maintained reliably using a service like Zookeeper [18].

As shown in Figure 1, clients communicate with servers to perform B-tree operations including search (contains), get (read), put (insert/update), and delete. Of these operations, search and get may be performed by clients via RDMA. Servers also communicate with each other to grow and maintain the distributed B-tree. The coordination between servers adds a level of complexity not present in prior RDMA-optimized systems like FaRM [7] or Pilaf [32]. However, we minimize this complexity by carefully designing our B-tree, discussed next.

2.2 Server-Side B-tree Operations

Cell uses a type of external B-tree called a *B-link tree* [25]. We use the same structure at both the meganode scope and within each meganode, as illustrated in Figure 2. B-link trees offer much higher concurrency than standard B-trees due to two structural differences: each level of the tree is connected by right-link pointers, and each node stores a *max key* which serves as an upper bound on the keys in its subtree. (We also store a min key to cope with concurrent RDMA reads; see 2.3.) Sagiv [38] refined the work of Lehman and Yao [25] into an algorithm that performs searches lock-free, insertions by locking at most one node at a time, and deletions by locking only one node. The lack of simultaneous locking makes the algorithm well-suited to distributed and concurrent settings [19, 30].

We follow Sagiv's algorithm when operating within a meganode. A search for a key follows child pointers and, if necessary, right-link pointers until the correct leaf node is reached. Range queries are implemented by following right links at the leaf level. Insertions and deletions begin with a search to find the correct leaf node. If an insertion causes a leaf node L to split (because it is full), we lock L long enough to create a new node L' containing roughly half of L's contents, and set L's right-link pointer to L'. The right link ensures that concurrent searches can reach L' (guided by the max key of L) even if it has no parent yet. The split key is then inserted into the parent as a separate, decoupled operation that may propagate further up the tree. Deletions simply remove the key from its leaf node under a lock. We avoid multi-lock compaction schemes and deletion rebalancing to improve concurrency [35, 12]; this practice has been shown to have provably good worst-case properties [40]. To limit overhead from underfilled leaf nodes, a desired ratio of tree size to total key-value data can be selected. The tree can then be periodically rebuilt offline without sacrificing liveness, by checkpointing the tree, rebuilding it with insertions into an empty tree, then replaying a delta of operations performed since the checkpoint was recorded and swapping the live and offline trees.

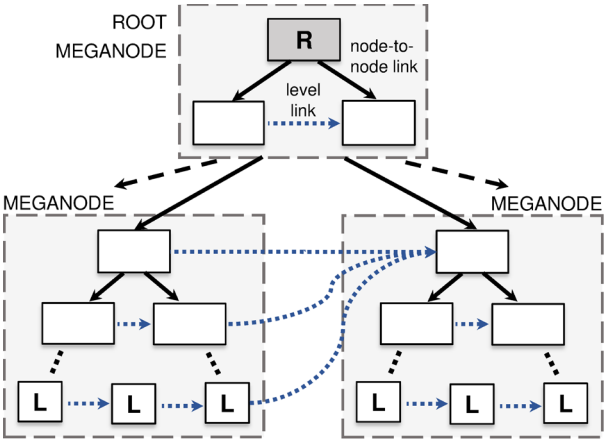


Figure 2: The structure of Cell's data store, a B-link tree of B-link trees. Each individual meganode contains a complete level-linked tree; the meganodes are also level-linked. The root meganode contains the root node (R). The leaves (L) of the bottom meganodes point to key-value data stored in the local extents region of that machine.

We extend Sagiv's algorithm to server-side operations in our meganode structure.

(Server-side) search and caching: To search for a key-value entry, clients iteratively traverse the tree one meganode at a time by sending search requests to the appropriate servers, starting at the server containing the root node R (Figure 2). Each server uses Sagiv's algorithm to search within meganode(s) until it reaches a pointer that is not local to the machine. This remote pointer is returned to the client, which continues the search request at the pointer target's server. When a leaf node is reached, the server returns the key-value pair from its extents region to the client. To bootstrap the

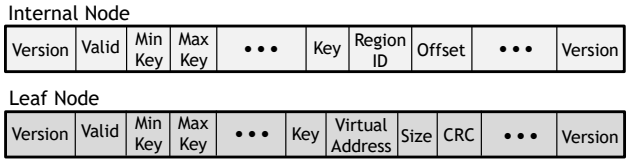


Figure 3: The structure of internal and leaf meganodes in Cell's B-link trees. Each node packs two matching versions, a minimum and maximum key, and zero or more pointers to other nodes or to extents memory in a block of a few kilobytes.

search, we ensure that a pointer to R is always stored at offset 0 in the region with the lowest ID across the cluster. We can speed up searches by having clients cache the depth and key range of meganodes close to the root because servers provide this metadata with traversal results. This type of caching is effective because updates in a B-tree such as ours occur exponentially infrequently in node height [40].

Deletion: To delete a key-value entry, a client orchestrates a server-side search for the key, then instructs the server to delete a leaf node entry according to Sagiv's algorithm. Nodes are not compacted to avoid multi-lock algorithms and maintain concurrency.

Insertion: To insert (or update) a key-value entry, a client performs a server-side search for the key, then instructs the server to insert a leaf node entry according to Sagiv's algorithm. As individual nodes fill and split (potentially propagating upwards) the meganode itself may become full, requiring that it split.

In principle, we could apply Sagiv's algorithm at the meganode level as well, but this would require locking the entire meganode for the duration of the split, blocking all other operations. Instead, we use a finer protocol inspired by Sagiv's algorithm that allows greater concurrency. It identifies a split key in a meganode X that divides it into two halves, X_{left} and X_{right}. X_{right} is locked for the duration of the split, but updates can continue in X_{left}. The server copies the nodes in X_{right} to a new local or remote meganode asynchronously[2]. Then, it locks X_{left} long enough to update the right-link pointers of X_{left} along the split boundary to point to the root of the new meganode. At this point, the meganode structure is restored as in Figure 2. Lastly, the server invalidates the old X_{right} by setting a boolean flag in each node, indicating that the nodes can be reused, and releases the lock on X_{right}'s key range.

A meganode should be split before it becomes too full, otherwise concurrent updates to X_{left} may fail if we run out of space. Note that client-side searches may occur throughout the meganode split process. Ensuring their correctness is subtle, as we discuss in Section 2.3.

[2]To balance network costs with the desire to balance the tree across all available servers, Cell favors remote meganodes when few local meganodes have been used, and local meganodes otherwise.

2.3 Client-Side Search and Caching

Cell organizes each meganode as a local B-link tree in order to enable client-side searches using RDMA reads. The search process is similar to the server-side equivalent, except that the client needs to iteratively fetch each B-tree node using an RDMA read, following child and right-link pointers, as it traverses a meganode. When the search terminates at a leaf node, the client attempts one additional RDMA read to fetch the actual key-value data from the server's extent region, or issues an insert or delete RPC to the server containing the leaf node.

A full-sized 64MB meganode built from 1KB-sized nodes contains a 5-level local B-link tree. Thus, RDMA search through a meganode takes up to 5 RTTs while server-side search requires only one. This is not as bad as it seems, because: (1) RDMA-enabled networks have very low RTTs, so the overall client-side search latency remains small despite the extra roundtrips. (2) The latency overhead pales in comparison to the queuing delay if the server CPU is bottlenecked at high load. To reduce search roundtrips and ameliorate hotspots at the root meganode, clients cache fetched nodes. Clients follow the same strategy as for server-side search: only cache nodes near the root, and if they follow a right-link pointer, they invalidate any information cached from the parent node.

Allowing client-side RDMA reads during server-side tree modifications introduces subtle concurrency challenges, as Sagiv's algorithm requires that individual nodes reads and writes appear atomic. This is easy to enforce among server-side operations using CPU synchronization primitives. However, no universal primitives exist to synchronize RDMA reads with the server's memory access. To ensure that RDMA reads are consistent, we use two techniques (see Figure 3):

- We store a version number at the start and end of each node. To update a node, the server increments both version numbers to the same odd value, performs the update, then increments both numbers to the same even value. Each step is followed by a memory barrier to flush the cache to main memory. If the RDMA read of a node is interleaved with the server's modification of that node, it will either see mismatched version numbers or the same odd value, indicating that the read must be retried. This method works because in practice RDMAs are performed by the NIC in increasing address order[3] and because individual nodes have fixed boundaries and fixed version number offsets.
- Key-value entries are variable size and have no fixed boundaries. We use the technique proposed in Pilaf [32] of storing a CRC over the key-value entry in

[3]To handle NICs that do not read in increasing address order, we need to adopt the solution of FaRM [7] of using a version number per cacheline.

the corresponding leaf node pointer. After performing an RDMA in the extent region, the client checks if the CRC of the data matches the CRC in the pointer; if not, the RDMA is retried. Like node modifications, each key-value write is also followed by a memory barrier.

2.4 Correctness

Cell dictionary operations (search, insert, and delete) are linearizable [16] so that both server-side and client-side searches will be correct. We refer readers to §4.2.1 of Mitchell's PhD thesis [33] for the proof.

Theorem 1. *Every Cell operation on small nodes maps a good state (in which all previously-inserted data that has not yet been deleted can be reached via a valid traversal) to another good state. Therefore, by the give-up theorem [41], it is linearizable.*

If meganodes never split, we can leverage Sagiv's proof of correctness [38]. However, it requires that node reads and writes be atomic, as guaranteed by the design in the previous section. Therefore, any search through a Cell store for x, if it terminates, will terminate in the correct place: the node $n \mid x \in keyset(n)$, where $keyset(n)$ is the set of keys that are stored in n. Since Cell satisfies the give-up theorem [41], and all Cell operations map good states to good states, Cell is linearizable.

The linearizability of operations during meganode splits is delicate, and requires an additional proof, also provided in Mitchell's thesis [33].

Theorem 2. *Insert, delete, and search operations remain linearizable during a meganode split. Insert and delete operations are blocked during the meganode split, and proceed correctly once the split is complete. Search operations can either continue or backtrack correctly during and after the split. Thus, Cell operations remain linearizable, by Theorem 1.*

The state stored in a node n allow multiple copies of a node to safely temporarily exist. Invalidating all but one copy of the node before allowing modifications to any copy of n removes any possible ambiguity in the set of keys present in Cell. The design for this is presented below. All Cell operations still map good states to good states as from Theorem 1. Thus, any search through a Cell store for x, if it terminates, will terminate in the correct node $n \mid x \in keyset(n)$, even during meganode splits.

Although the server can block its own write operations during a meganode split, it cannot block clients' RDMA reads, so we must ensure the latter remain correct. The problem occurs during node invalidation and reuse. By resetting the valid bit in each node in X_{right}, the server guarantees that concurrent client-side searches fail upon reading an invalid node. However, an invalid node might be reused immediately and inserted into an arbitrary location in X_{left}. A client-side search intending to traverse X_{right} might read this newly reincarnated node instead[4]. The min and max keys in each node allows a client to detect this case and restart the search.

2.5 Selectively Relaxed Locality

Traditional systems wisdom encourages maximizing locality between data and computation over that data to reduce expensive data copying over the network. RDMA greatly reduces the cost of moving data to computation, so Cell's hierarchical B-tree design allows for both server-side searches (over local data) and RDMA searches (over data copied from a Cell server). When servers are under low load and/or client resources are scarce, server-side searches achieve better overall resource efficiency. However, clients should switch to using RDMA searches when servers become overloaded and client CPUs are available. How should clients (and servers performing cross-server data structure modification) dynamically relax locality, i.e., decide which search method to use?

To answer this question, we model the system using basic queuing theory [14]. Specifically, we model each Cell server as consisting of two queues, one (Q_s) for processing server-side searches, the other (Q_r) for processing RDMA read operations. The service capacity (ops/sec) of Q_s is T_s, which is determined by the server's CPU capability, and the service capacity of Q_r is T_r, which is determined by the NIC's RDMA read capacity. We assume that Q_s and Q_r are independent of each other.

Let q_s and q_r represent the current lengths of the queues, respectively. Since our job sizes are fixed, the optimal strategy is to Join the Shortest Queue (JSQ) [13, 48]. This decision is made on a per meganode basis. More concretely, after normalizing queue length by each queue's service time, a client should join Q_s if $\frac{q_s}{T_s} < \frac{q_r}{T_r}$, and Q_r otherwise. We need to make another adjustment when applying JSQ: since each RDMA search involves m RDMA reads, the client should choose server-side search if the following inequality holds:

$$\frac{q_s}{T_s} < m \times \frac{q_r}{T_r} \tag{1}$$

Instead of directly measuring the queue lengths (q_s and q_r), which is difficult to do, we examine two more easily measurable quantities, l_s and l_r. l_s denotes the latency of the search if it is done as a server-side operation. It includes both the queuing delay and round-trip latency, i.e. $l_s = \frac{q_s}{T_s} + RTT$. l_r denotes the latency of an RDMA read during an RDMA search, i.e. $l_r = \frac{q_r}{T_r} + RTT$. Substituting $\frac{q_s}{T_s} = l_s - RTT$ and $\frac{q_r}{T_r} = l_r - RTT$ into inequality (1) gives us the final search choice strategy.

[4] Because nodes have fixed boundaries, we do not need to worry that a client read might read in the middle of a node.

To determine inequality 1, we need to estimate various terms. For a 64MB meganode with 1KB nodes, we initially set $m = 5$, the estimated height of each meganode in nodes; as we traverse meganodes, we adjust this estimate based on the average meganode height. We set RTT to be the lowest measured RDMA latency to the server. We approximate the current server-side search latency (l_s) and RDMA latency (l_r) by their past measured values. Over the time scale that the queue estimation computations are being performed, the rates of queue filling and draining do not change dramatically, so the average queue length also remains relatively stable. A client performing continuous operations may get new latency measurements (i.e., queue length proxy measurements) as often as every $10\mu s$ to $50\mu s$.

Additionally, we apply the following refinements to improve the performance of our locality selector:

- *Coping with transient network conditions:* We avoid modeling short-term transient network conditions with two improvements. First, we use a moving average to estimate l_s and l_r: clients keep a history of the most recent n (e.g. 100) samples for each server connection and calculate the averages. Second, we discard any outlier sample s if $|s - \mu| \geq K\sigma$, where μ and σ are the moving average and standard deviation, respectively, and K is a constant (e.g. 3). If we discard more samples than we keep in one moving period, we discard that connection's history of samples.
- *Improving freshness of estimates:* We perform randomized exploration to discover changes in the environment. With a small probability p (e.g., 1%), we choose the method estimated to be worse for a given search to see if conditions have changed. If a client has not performed any searches on a connection for long enough (e.g., 3 seconds), that connection's history is discarded.

The constants suggested above were experimentally determined to be effective on a range of InfiniBand NICs and under various network and CPU loads. We found that the performance of the selector was not very sensitive to changes in these values, as long as $K > 1$, p is small, and n covers timescales from milliseconds to one second.

2.6 Failure Recovery

Cell servers log all writes to B-tree nodes and key-value extents to per-region log files stored in reliable storage. The log storage should be replicated and accessible from the network, e.g. Amazon's Elastic Block Store (EBS) or HDFS. Our prototype implementation simply logs to servers' local disks. When a server S fails, the remaining servers split the responsibility of S and take over its memory regions in parallel by recovering meganodes and key-value extents from the corresponding logs of those regions. No remote pointers in the B-link tree need to be updated because they only store region IDs; the external (e.g., ZooKeeper-stored) map of region IDs to physical servers suffices to direct searches to the correct server.

Operation logging is largely straightforward, with one exception. During a meganode split, server S first creates and populates a new meganode before changing the right links of existing nodes to point to the new meganode. If S fails between these two steps, the new meganode is orphaned. To ensure orphans are properly deleted, servers log the start and completion of each meganode split and check for orphaned meganodes upon finding unfinished splits in the log. Node splits are handled similarly.

3 Implementation

We implemented Cell in ~18,000 lines of C++. Cell uses the `libibverbs` library, which allows user-space processes to use InfiniBand's RDMA and message-passing primitives using functions called *verbs*. We use Reliable Connection (RC) as the transport for both RDMA reads and SEND/RECV verbs. We use SEND/RECV verbs to create a simple RPC layer with messaging passing, used for client-server and server-server messages. Although client-side search requires that the client and server both have logic to traverse the tree, this code can be reused to limit the additional implementation effort.

Our server implementation is single-threaded; the polling thread also processes the Cell requests and performs server-server interactions for meganode splits. Like HERD [22], we run multiple server processes per machine in order to take advantage of multiple CPU cores. As suggested by FaRM's [7] findings on combining connections, we implemented a multi-threaded Cell client, and experimentally chose 3 threads per client process for most tests. To further increase parallelism, the client supports pipelined operations to avoid idling by keeping multiple key-value operations outstanding.

Client-side searches fetch 1KB nodes via RDMA to traverse a meganode. Server-side searches involve sending an RPC request to traverse a given meganode, and receiving the pointer to the meganode at the next meganode level along the path to that key's leaf node.

Clients cache B-link tree nodes to accelerate future traversals, maintaining an LRU cache of up to 128MB of 1KB nodes. We only cache nodes at least four node levels above the leaf node level to minimize churn and maximize hits. Symmetrically, each client maintains an LRU cache of up to 4K server-side traversal paths leading to the leaf-level meganodes, indexed by the key range covered by that meganode. We used in-band metadata to allow clients to verify the integrity of all RDMA reads. For the CRCs covering extents, we use 64-bit CRCs to make the probability of collisions vanishingly small. B-link tree nodes are protected by the previously discussed low and high version numbers.

Cell servers pre-allocate large pools of memory for B-link tree nodes and key-value extents. The extents pool is

managed by our own version of SQLite's `mem5` memory manager. We support `hugetlbfs` backing for these large memory pools to reduce page table cache churn [7].

Node size and network amplification: We choose 1KB nodes for most of our evaluation. This number is not arbitrary: in our setup, 1KB RDMA fetches represent the point above which RDMA latency goes from being flat to growing linearly, and throughput switches from ops/sec-bound to bandwidth-bound.

RDMA-based traversals do incur a significant bandwidth amplification over server-side searches, on the order of $4{\cdot}1\text{KB}/64\text{ bytes} = 64\times$ with caching enabled. Using smaller nodes would shift the balance between bandwidth amplification and traversal time: for example, 512-byte nodes would add 4 levels (33%) to a 10^{15}-key tree while only enabling 16% more node fetches/sec per NIC. Client-side traversals would require 33% less bandwidth, but would take 12% longer.

4 Evaluation

We evaluated Cell's performance and scalability on the PRObE Nome cluster [1]. The highlights of the results:

- Cell adapts to server CPU and network resources across configurations. When the server CPU is more bottlenecked than its NIC (e.g. using a single core at the server), Cell achieves 439K searches/sec while server-side-only achieves 164K searches/sec. When the server is configured with 8 cores per server, Cell mostly uses server-side-processing, achieving 1.1 million ops/sec, 7% better than server-side-only and $3.7\times$ better than client-side-only.
- Cell handles load spikes that cause transient server CPU bottlenecks and increased queuing delay by increasing the ratio of client-side processing.
- Cell scales to 5.31 million search ops/sec using 16 Cell servers, with 2 cores each.
- Cell is effective for any mix of B-tree get and put operations, including those that cause online tree growth.

4.1 Experimental Setup

Hardware and configuration: Our experiments were performed on the PRObE Nome cluster. Each machine is equipped with 4 quad-core AMD processors and 32GB of memory, as well as a Mellanox MHGH28-XTC ConnectX EN DDR 20Gbps InfiniBand NIC and two Intel gigabit Ethernet adapters. Cell was run on top of CentOS 6.5 with the OFED 2.4 InfiniBand drivers.

For each experiment, we use distinct sets of server and client machines. Unless otherwise specified, we use:

- 4 server machines with 2 cores per server (by running 2 server processes per machine)
- the remaining machines in each experiment for clients
- 20M key-value pairs populated per server

Throughput at saturation and latency at a moderate (non-saturation) load are reported unless otherwise specified.

We enable `hugetlbfs` support on our server machines so that the RDMA-readable node and extents data can be placed in 1GB hugepages. Due to the complexity of modifying the InfiniBand drivers, we do not attempt to put connection state in hugepages, as our experiments indicate this would yield minimal impact on performance at scale due to other sources of InfiniBand latency [7].

We allow clients to consume up to 128MB of RAM to cache B-link tree nodes. To approximate performance with a much larger tree, we prevent the bottom four node levels of the tree from being cached, effectively limiting the cache to the top three levels in most of our tests.

Cell's throughput does not decrease when synchronous logging is enabled for key-value sizes below 500 bytes. For 100% put workloads that insert key-value pairs larger than 500 bytes, the I/O bandwidth of the SSD in each of our local cluster's servers becomes a bottleneck. As Nome's machines lack SSDs, we disable Cell's asynchronous logging in our experiments.

Workloads: To test search, get, and put (insert/update) operations, we generate random keys uniformly distributed from 8 to 64 characters, and values from 8 to 256 characters. We focus on evaluating the performance of search, as that is the dominant operation in any workload. For real-world benchmarks, we also utilize the Zipfian-distributed YCSB-A (50% puts, 50% gets) and YCSB-B (5% puts, 95% gets) workloads. All other tests use keys selected with uniform probability.

4.2 Microbenchmarks

Systems with relaxed locality are fast. Operating on a single server machine, Cell surpass the latency and throughput of schemes that maintain strict locality. This section evaluates Cell's search performance on a single meganode on a single CPU core without caching, using server-side search, client-side search, and selectively relaxed locality. Cell's performance on several cores per single machine is also measured.

Raw InfiniBand operations: We measure the throughput and latency of 1KB RDMA reads and 128-byte two-way Verb ping-pongs on 1 to 16 servers, utilizing 1 CPU core per machine (Table 1). Because very little per-message processing is performed on the servers, the CPU is not a bottleneck. We vary the client count to search the throughput-latency space for RDMA reads, Verb messaging, and simultaneous use of both. Because latency rises with no additional throughput past saturation, we report the throughput with the lowest latency within 5% of the maximum measured throughput.

Previous work has demonstrated higher throughput for RDMA and Verb messaging [22, 7]. FaRM reported $4\times$ 1KB RDMA read throughput on newer, $2\times$ throughput NICs [7]. We attempted to replicate HERD's published results using their benchmarks on the PRObE Susitna

testbed, equipped with 40Gbps InfiniBand cards. We reached 16M ops/sec for one server with 4-byte RDMA reads accessing the *same* 4 bytes repeatedly within a 5MB region, compared with 20M ops/sec reported by the authors [22]. However, when we switched to 1KB RDMA reads spread through the entire 5MB region, the throughput dropped to 1.69M ops/sec, and further enlarging the RDMA-readable region to a more realistic 8GB dropped the throughput to 1.3M ops/sec. Notably, the tests throughout the remainder of this evaluation were run on older 20Gbps InfiniBand cards that are bottlenecked at a similar operation limit, 1.04M ops/sec over a 1GB area (or 1.44M ops/sec over a 64MB area).

Similarly, we tested HERD's Verb benchmarks. We achieved 12M ops/sec for 16-byte Verb messages exchanged over the Unreliable Connection (UC) transport, and 8.5M ops/sec for 128-byte messages over RC. In the HERD benchmark, each client only communicates with a single server. With one-to-one communication and a very small number of connections (10), we were able to achieve similar performance with our own benchmarks. Clients in a real-world distributed storage system need to engage in all-to-all communication with all servers, so our microbenchmarks (Table 1) report results with all-to-all communication with up to 100 clients.

We emphasize that selective locality can balance CPU and network, adapting to the available resources and providing better *relative* results than server-side or client-side processing alone. Some of our experimental conditions are more realistic than previous work, while we omit some difficult (but reasonable) optimizations that prior work uses. While our raw InfiniBand results differ in absolute numbers, Cell is able to balance the CPU and network resources at the level appropriate for a given environment, including those with better network performance and/or more CPU resources than our prototype.

Search throughput with a single meganode: To establish the baseline performance of Cell's selectively relaxed locality approach, we present a microbenchmark for traversals of a single meganode served by 1 server CPU core. We examine the performance of client-side only searches, server-side only searches, and Cell's locality selector, each with client-side caching disabled. Figure 4 demonstrates the throughput-latency relationship as we increase the number of client processes from 1 to 12, distributed across 4 client machines. With the server performing B-tree searches, 1 CPU core is no longer sufficient to saturate the NIC; the server-side search's peak throughput is 158K searches/sec, and the bottleneck is the server CPU. Client-side-only search achieves 305K search/sec; since 5 RDMA reads are required to traverse the 5-level meganode used and caching is disabled, this matches our raw InfiniBand performance of 1.44M ops/sec over a 64MB region. Combining the two methods yields 93% of the *aggregate* throughput, peaking at 432K searches/sec. For all except the 1-client (lowest-throughput) point on each line, the server's single CPU core is already saturated with the number of operations it can perform per second. RDMA provides Cell with extra throughput, and in this case lower latency, because the extra latency from multiple RDMA round trips is overshadowed by the queuing delay at the server for the server-side case.

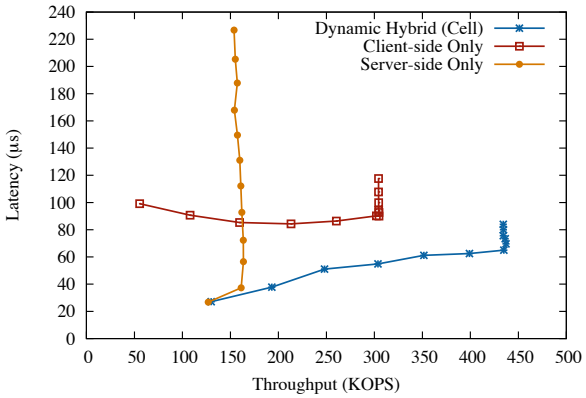


Figure 4: Throughput and latency for server-side search, client-side search, and Cell's locality selector. The experiments involve 1 server serving one meganode, utilizing one core. Client B-tree caching and operation pipelining are disabled.

To calibrate Cell's local in-memory B-tree implementation, we compare its performance and MassTree's [31]. Using key and value distributions matching Cell's and the `jemalloc` allocator, we measure MassTree's performance at 276K local B-tree searches/sec (using a single core), compared to Cell's 379K local searches/sec. This suggest Cell's local B-tree implementation is competitive. When adding the cost of network communication, Cell performs 158K server-side searches/sec. MassTree does not have InfiniBand support, nor does it implement a distributed B-tree. Thus, we do not compare with MassTree further.

4.3 Performance at Scale

Cell scales well across many servers. We vary the number of server machines from 1 to 16, using 2 CPU cores per server and enabling client caching. We scale the size of the B-tree to the number of servers, at 20M tuples per server, storing up to 320M tuples for 16 servers. We also use enough clients to saturate each set of Cell servers. Our biggest experiments consist of 64 machines (16 servers plus 48 clients) with 2560 client-server connections (80 3-threaded client processes to saturate the 32 server cores). Our largest B-tree is 2 meganodes tall and stores 320M tuples.

Throughput: Figure 5 shows the search throughput in logscale of Cell as well as the alternative of server-side processing only. It demonstrates that Cell displays near-linear throughput scaling over additional servers (satu-

	RDMA read		Verb messaging		Hybrid	
Servers	Throughput	Latency	Throughput	Latency	Throughput	Latency
1	1.04M ops	15.5μs	750K ops	21.4μs	1.60M ops	20.1μs
4	4.13M ops	15.5μs	2.96M ops	21.7μs	6.00M ops	21.8μs
8	8.77M ops	14.6μs	5.88M ops	21.5μs	10.58M ops	20.2μs
24	24.97M ops	15.5μs	18.15M ops	22.5μs	35.56M ops	22.0μs

Table 1: Microbenchmarks of throughput and latency at maximum throughput for 1KB RDMA reads over a 1GB area, 128-byte 2-way Verb messages, and both. All reported values are within 5% of the peak measured, at minimum latency.

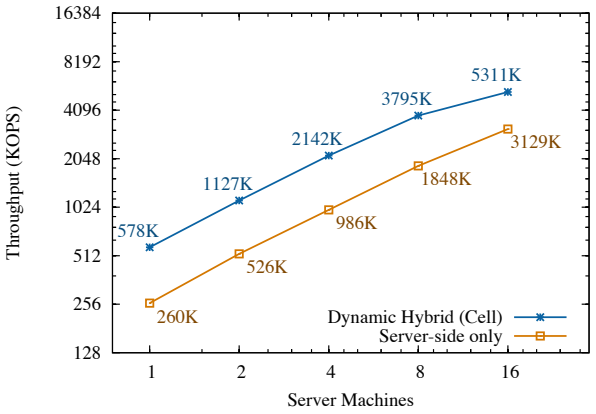


Figure 5: Throughput as the number of servers increases from 1 to 16. Each server uses 2 CPU cores. 4 client machines per server are used to saturate the servers.

rated with additional clients) until queue pair state overwhelms the NICs' onboard memory. Cell's throughput increases 9.2× from 1 server to 16 servers. Compared to the server-side only approach, Cell's hybrid approach achieves 70% higher throughput with 16 servers.

Latency: The B-tree remains at 2 meganode levels and 7 node levels as the number of servers grows from 4 to 16, so the overall search latency is stable. Below saturation, the median latency for both the server-side only approach and Cell is ~30μs.

4.4 Locality Selector: Balancing CPU and Network

Cell's locality selector effectively chooses the correct search method to use under arbitrary network and server load conditions. The selector is designed to minimize operation latency while rationing server CPU resources. We evaluate its performance by asking three questions:

1. How effectively does Cell use CPU resources?
2. How accurately does Cell estimate and balance search costs for a given environment?
3. When is the locality selector beneficial?

This section answers all three questions.

4.4.1 Varying Server CPU Resource

We measure the performance of Cell and server-side search from 1 to 8 server CPU cores on 1 server machine and 4 server machines. Table 2 compares Cell's locality selector and 100% server-side search on 4 servers, showing that Cell is able to consistently match or exceed server-side search's throughput. With 2 cores, Cell achieves 74% of server-side search's maximum throughput, for which the latter requires 7 cores. Cell's selective locality approach allows the system to dynamically balance server CPU, client CPU, and network bandwidth usage, so a Cell storage cluster can indeed economically satisfy transient peak usage with fewer CPU cores. In addition, in applications where server-side operations are more CPU-intensive and each CPU core can therefore complete fewer operations each second, Cell's advantage becomes even more pronounced.

Table 2 indicates that Cell's hybrid scheme can extract 2.13M searches per second from 2 CPU cores on each of 4 server machines, more than double the 991K server-side only searches at the same CPU count. Server-side searches are able to reach 2.90M searches per second using 7 cores per machine, close to the theoretical 3.0M ops/sec maximum Verb throughput our microbenchmarks suggest. This 3.5× resource expenditure yields only 35% higher throughput compared with the hybrid selector on 2 cores per machines, for example. The latency of hybrid operations remains consistently low even with Cell utilizing few CPU cores; with 2 cores and moderate load, searches average 28.5μs, dropping to 26.4μs with 4 cores per server. Although modern servers typically have many more than 2 cores, even in a dedicated environment, cores no longer needed for storage logic can be devoted to CPU-intensive tasks data, including summarization, aggregation, analysis, cryptographic verification, and more. The server-side-only advantage in Table 2 for $\geq$ 7 cores is due to the fact that although only 1% of requests are performed client-side, with plentiful CPU resources the latency of client-side requests is significantly higher than server-side requests.

If CPU cores are plentiful, Cell has the same behavior as server-side processing and its performance is bounded by the InfiniBand IOPs. Using the Unreliable Datagram (UD) transport instead of the Reliable Connection (RC) transport can be used to avoid accumulating connection state and thus greatly increase messaging performance [24]. With UD, the higher messaging throughput can be saturated with additional server CPUs, if available, but RDMA is not available to provide load balancing. In a shared cluster with finite CPU resources handling a complex data structure like a distributed B-tree, the extra UD capacity would likely go to waste.

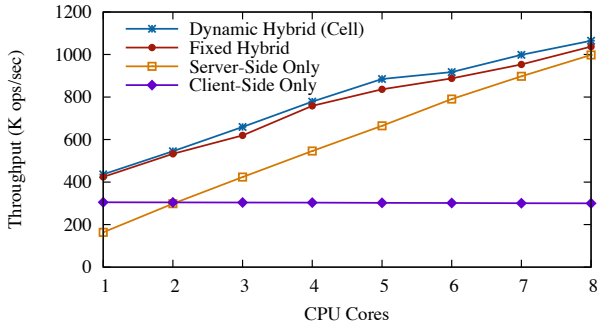


Figure 6: Throughput of 1 server running Cell on 1 to 8 CPU cores, using Cell's locality selector, a manually-tuned percentage of server-side operations, server-side only, and client-side only operations.

4.4.2 Dynamic Cost Estimation and Balancing

We find that under a constant server load, there is a per-client fixed ratio of server-side and client-side searches that produces maximal throughput at minimal latency. However, as illustrated in Table 2, these ratios shift dramatically as the available server resources change, and must be re-tuned for every change in the number of available cores per server, the number of servers, the number of clients, the workload, and the request rate. Cell correctly picks ratios that yield minimum median latency regardless of the server load; in fact, because the ideal ratio is slightly different for each client due to network topology and the position of its connection in the NIC's state, we observe Cell picking different ratios on each client that produce globally-optimized throughput and latency. Figure 6 shows that in most cases, especially when few resources are available, Cell meets or exceeds the throughput of the best hand-tuned fixed ratio, as it can continuously adjust for current load conditions.

Other environments: Tests on two other clusters with different NICs and CPUs yielded similar results. With similar 2-server-CPU experiments, Cell adapted with different ratios of client- to server-side search, using more client-side searches on Susitna's powerful NICs [1] and more server-side searches on our local cluster.

4.4.3 Load Spikes and Load Balancing

Cell's ability to maintain low latency in the face of transient server load demonstrates the value of dynamically selecting between RDMA and messaging-based traversal. In the absence of applicable traces from real-world systems at scale, we perform microbenchmarks of Cell's ability to rapidly adapt to load spikes and workload changes. Figure 7 compares the application-facing latency of Cell clients and clients that use only server-side search when the load on a cluster of 4 servers increases unexpectedly for 5 seconds. In these tests, application search requests arrive at each of 24 Cell clients every $75\mu s$. For 5 seconds, the load rises to $2.5\times$ as an additional 36 clients begin injecting searches at the same rate. Figure 7 shows that Cell is able to very rapidly switch the majority of its searches to client-side traversals, maintaining low median and 90th percentile latencies compared to server-side search. Cell's locality selector effectively manages server CPU resources to minimize latency in the face of long-term and short-term changes in server load and available resources.

Cores	Server-Side Only	Cell	Fixed Ratio
1	505K	1842K (30%)	1841K
2	991K	2142K (41%)	2129K
3	1447K	2456K (53%)	2366K
4	1831K	2573K (62%)	2513K
5	2145K	2687K (77%)	2593K
6	2478K	2845K (88%)	2768K
7	2901K	2901K (99%)	2776K
8	2543K	2812K (99%)	2571K

Table 2: Search throughput of 4 server machines utilizing 1 to 8 CPUs per server: 100% server-side searches, Cell (with the average of its dynamically-selected ratio of server-side searches), and a fixed percentage of server side searches using that average. On 8 or more CPUs, the amount of connection state necessary to connect servers to clients causes predictable degradation. Cell saturates with fewer clients, so this effect is less pronounced.

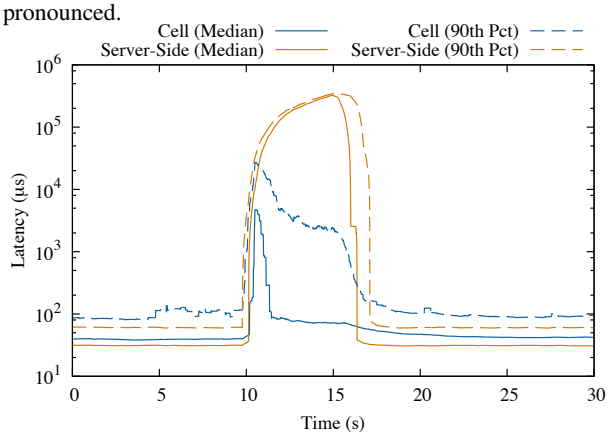


Figure 7: Request latency during 5-second transient load spike. These tests were performed with client pipelining disabled for simplicity. Note logarithmic y-axis.

4.5 Read-Write Concurrency

Like most sorted stores, Cell can perform search, get, put, delete, and range operations. We benchmark mixes of get and put operations to ensure that Cell retains its advantages beyond search operations.

Cell is designed to maintain low latency and to support concurrent read operations even when servers are modifying the tree state, for puts or for node or meganode splits. Notably, the locality selector optimizes for server utilization rather than allowing any single client to optimize for its own latency. Figure 8 traces the median latency of a group of 8 clients performing searches while a separate group of 16 clients idles, then performs bulk

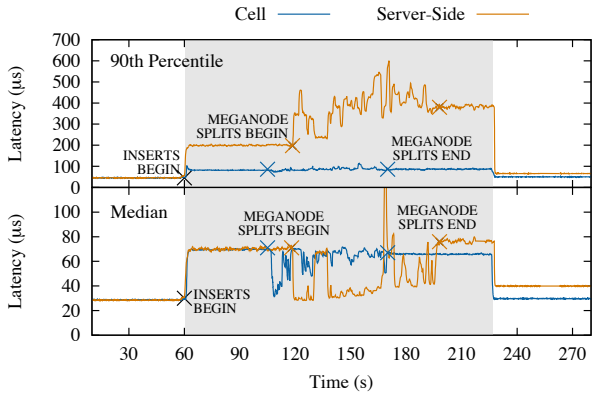


Figure 8: Latency of reads on 8 clients (24 threads) without and with 16 clients (48 threads) performing continuous insert operations on a cluster of 4 servers running 2 Cell threads each.

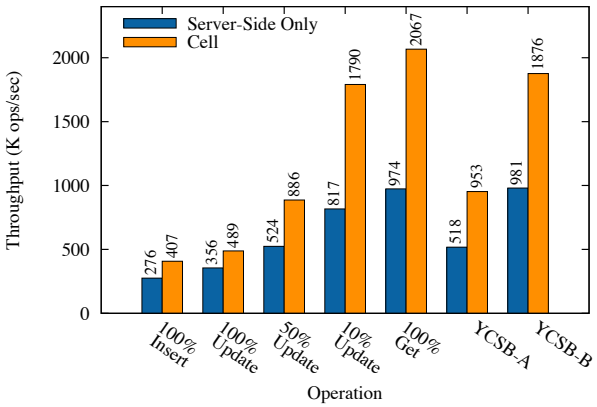


Figure 9: Throughput of pipelined mixed Get and Put workloads on 4 servers, 2 CPUs per server.

put operations that cause the tree structure to grow. With Cell, the search clients maintain concurrency by using more client-side operations, so a set of meganode splits completes faster. The median latency of Cell searches is thus slightly higher than the server-side equivalent, while the 90th percentile latency is far lower. Server-side search latency is impacted more significantly by write operations as the server's CPU capacity is shared between the two. With Cell we are able to dynamically shift the search operations to the client, incurring a slightly higher latency over server-side search, while using the saved server CPU cycles to execute node and meganode split operations with reduced latency.

Therefore, workloads that combine read and write operations can maintain lower latency (and higher throughput) with Cell than with server-side only operations. We test mixes of get (rather than search) and update operations from 100% get to 100% update, as well as 100% insert and two YCSB benchmarks. Figure 9 shows that Cell consistently outperforms server-side only operations across a range of workloads. We do not report the performance of range (range queries) because for queries that return a single key, range has the same performance as get. As the set of keys returned by range grows, the per-key range performance improves, as up to a full leaf-level node of keys can be returned by each RDMA fetch within the range operation.

5 Related Work

There are two general approaches to using RDMA in distributed systems. One is to use RDMA to improve the throughput of the message-passing substrate of a system, increasing the overall performance when the system is bottlenecked by its message-passing capability. We call this approach RDMA-optimized communication. The other approach is to use RDMA to bypass servers' CPUs to improve the performance of systems bottlenecked by that resource. We refer to this approach as systems with client-side processing. In this section, we discuss related projects exploring both approaches as well as work on distributed storage.

RDMA-optimized communication: The HPC community has exploited the performance advantage of RDMA extensively to improve the MPI communication substrate [28, 27, 42]. The systems community has recently begun to explore the use of RDMA in distributed systems. Most projects focus on using RDMA to improve the underlying message-passing substrate of systems such as in-memory key-value caches [22, 21, 20, 44, 23], HBase [17], Hadoop [29], PVFSpvfs and NFS [11]. For example, HERD has proposed using an RDMA write to send a request to a server and to have the server respond using an unreliable datagram (UD) message. FaRM [7, 8] also uses RDMA writes to implement a fast message passing primitive. The advantage of this approach is that the resulting solutions are generally applicable, because all distributed systems could use a high performance message-passing primitive for communication. However, when the server's CPU becomes the bottleneck instead of the network, this approach does not take full advantage of RDMA to relieve servers' CPU load.

Systems with client-side processing: Several recent systems exploit RDMA's CPU-bypassing capability. Pilaf [32] builds a distributed key-value store which uses client-side processing for hash table lookups. FaRM [7, 9] provides a distributed transactional in-memory chunk store which processes read-only transactions at the client. We could potentially layer Cell's design on top of FaRM, but this would require using distributed transactions to modify the B-tree [4, 43]. With Cell's approach, we do not need distributed transactions (Section 2). DrTM [47, 6] offers distributed transactions over RDMA by exploiting hardware transactional memory (HTM) support.

Cell is inspired by FaRM and Pilaf to offload the pro-

cessing of read-only requests to the clients via RDMA reads. The main difference between Cell and prior work is that Cell explicitly tries to balance the server CPU and network bottleneck by carefully choosing when to prefer client-side over server-side processing.

Distributed B-trees: There are two high-level approaches for constructing a distributed B-tree. One builds a distributed tree out of lean (1-8KB) nodes [30, 4, 43]. Small nodes are crucial in the design of these systems, because the systems are all built on top of a distributed chunk/storage layer, and clients traverse the tree by fetching entire nodes over the network. The other approach, pioneered by Google's BigTable [5], builds a tree out of fat nodes (like meganodes; up to 200MB) [15]. This design reduces the number of different machines that each search needs to contact, but requires server-side processing to search within a meganode. Cell combines the two approaches.

Cell also handles concurrency differently than prior systems. Both Johnson and Colbrook [19] and Boxwood [30] implement distributed B-trees based on Sagiv's B-link tree. Johnson and Colbrook doubly-link the levels of the tree, merge nodes on deletion, and cache internal nodes consistently across servers, resulting in a complex scheme that requires distributed locks. Cell uses only local locks, similar to Boxwood. Furthermore, our caching of internal nodes is only advisory in that stale caches do not affect the correctness of the search. Aguilera et al. [4] implement a regular B-tree and handle concurrency requirements using distributed transactions, which are more conservative than necessary and are especially heavy-handed for read-only searches. Minuet [43] expands on Aguilera et al.[4], addressing some of the scalability bottlenecks and adding multiversioning and consistent snapshots. Some of their improvements emulate the B-link tree, yet they do not seem to benefit from the simplicity of Sagiv's single-locking scheme.

Other in-memory distributed storage: The high latency of disk-based storage has led to a large research effort behind in-memory storage. Memcached [10] and Redis [39] are popular open source distributed key-value stores. The RAMCloud project explores novel failure recovery mechanisms in an in-memory key-value store [36, 37], although it does not take particular advantage of RDMA. Masstree [31] and Silo [45] provide fast single-server in-memory B-tree implementations. MICA [26] presents a fast single-machine implementation of MassTree. These are not distributed and are not based on B-link trees.

6 Conclusion

RDMA opens up a new design space for distributed systems where clients can process some requests by fetching the server's state without involving its CPU. Mixing client-side and server-side processing allows a system to adapt to the available resources and current workload. Cell achieves up to 5.31M search ops/sec on 32 CPU cores across 16 servers, at an unsaturated latency of $\sim 30\mu s$. Cell saves up to 3 CPU cores per server per InfiniBand NIC, freeing resources for other CPU-intensive application logic, and maintains high throughput and low latency in the face of load spikes.

In looking towards Cell as the backend for real databases, we experimented with transactions and other database features. For example, RDMA can be used to quickly determine whether any item in a transaction's write set is currently locked by fetching many leaves across many servers simultaneously, and the status of locked keys can be re-checked with minimal resource consumption just by refetching that leaf.

From building and refining Cell, we learned lessons applicable to designing any distributed data store exploiting RDMA. Almost any RDMA-traversable distributed data structure that supports concurrent server-side writes can be built from CRC-protected variable-length data elements, version-protected fixed-length data elements, and a globally-known data structure root location. We learned that Pilaf's CRC approach works poorly for nodes in a structure with few roots, as each CRC update propagates to the root. Finally, an approach similar to our meganode split operation can make other types of distributed data structures' mutations friendly to concurrent RDMA access. Such mutations should replicate data, modify or rearrange data as necessary, then atomically (from the clients point of view) update links to old data to instead point to new data.

Acknowledgments

We would like to thank our shepherd, Peter Pietzuch, for his guidance and helpful suggestions. We appreciate our reviewers' insightful comments on this and earlier manuscripts that helped us hone this work.

Jinyang and Christopher were supported by NSF grant CNS-1409942.

References

[1] PRObE: testbeds for large-scale systems research. New Mexico Consortium and NSF and Carnegie Mellon University, `http://nmc-probe.org`.

[2] TritonSort: A balanced and energy-efficient large-scale sorting system. *ACM Transactions on Computer Systems 31*, 1 (2013).

[3] Extending high performance capabilities for Microsoft Azure, 2014.

[4] AGUILERA, M. K., GOLAB, W. M., AND SHAH, M. A. A practical scalable distributed B-tree. *Proc. VLDB Endow. 1*, 1 (2008), 598–609.

[5] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E.

[6] CHEN, Y., WEI, X., SHI, J., CHEN, R., AND CHEN, H. Fast and general distributed transactions using rdma and htm. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 26.

[7] DRAGOJEVIĆ, A., NARAYANAN, D., CASTRO, M., AND HODSON, O. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation* (2014).

[8] DRAGOJEVIĆ, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 54–70.

[9] DRAGOJEVIC, A., NARAYANAN, D., NIGHTINGALE, E. B., RENZELMANN, M., SHAMIS, A., BADAM, A., AND CASTRO, M. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of ACM Symposium on Operating Systems Principles* (2015).

[10] FITZPATRICK, B. Distributed caching with Memcached. *Linux J. 2004*, 124 (Aug. 2004), 5–.

[11] GIBSON, G., AND TANTISIRIROJ, W. Network File System (NFS) in high performance networks. Tech. rep., Carnegie Mellon University, 2008.

[12] GRAY, J., AND REUTER, A., Eds. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California, 1993.

[13] HAIGHT, F. Two queues in parallel. *Biometrika 45*, 3 (1958).

[14] HARCHOL-BALTER, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[15] HBase. `http://hbase.apache.org/`.

[16] HERLIHY, M., AND WING, J. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[17] HUANG, J., OUYANG, X., JOSE, J., UR RAHMAN, M. W., WANG, H., LUO, M., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High-performance design of HBase with RDMA over InfiniBand.

[18] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference* (2010).

[19] JOHNSON, T., AND COLBROOK, A. A distributed data-balanced dictionary based on the B-link tre. Tech. Rep. MIT/LCS/TR-530, Massachusetts Institute of Technology, 1992.

[20] JOSE, J., SUBRAMONI, H., KANDALLA, K., WASI-UR RAHMAN, M., WANG, H., NARRAVULA, S., AND PANDA, D. K. Scalable memcached design for InfiniBand clusters using hybrid transports. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (2012).

[21] JOSE, J., SUBRAMONI, H., LUO, M., ZHANG, M., HUANG, J., WASI-UR RAHMAN, M., ISLAM, N. S., OUYANG, X., WANG, H., SUR, S., AND PANDA, D. K. Memcached design on high performance RDMA capable interconnects. In *Proceedings of the 2011 International Conference on Parallel Processing* (2011).

[22] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 295–306.

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference* (2016).

[24] KOOP, M. J. *High-Performance Multi-Transport MPI Design for Ultra-Scale InfiniBand Clusters*. PhD thesis, The Ohio State University, 2009.

[25] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 6*, 4 (1981), 650–670.

[26] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014), 36.

[27] LIU, J., JIANG, W., WYCKOFF, P., PANDA, D., ASHTON, D., BUNTINAS, D., GROPP, W., AND TOONEN, B. Design and implementation of MPICH2 over InfiniBand with RDMA support. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (april 2004), p. 16.

[28] LIU, J., WU, J., KINI, S., BUNTINAS, D., YU, W., CHANDRASEKARAN, B., NORONHA, R., WYCKOFF, P., AND PANDA, D. MPI over InfiniBand: Early experiences. In *Ohio State University Technical Report* (2003).

[29] LU, X., ISLAM, N., WASI-UR RAHMAN, M., JOSE, J., SUBRAMONI, H., WANG, H., AND PANDA, D. High-performance design of Hadoop RPC with RDMA over InfiniBand. In *International Conference on Parallel Processing (ICPP)* (2013).

[30] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *USENIX Symposium on Operating System Design and Implementation* (2004).

[31] MAO, Y., KOHLER, E., AND MORRIS, R. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), pp. 183–196.

[32] MITCHELL, C., GENG, Y., AND LI, J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store. In *USENIX Annual Technical Conference* (2013).

[33] MITCHELL, C. R. *Building Fast, CPU-Efficient Distributed Systems on Ultra-Low Latency, RDMA-Capable Networks*. PhD thesis, Courant Institute of Mathematical Sciences, New York, 8 2015.

[34] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling Memcached at facebook. In *Proceedings of USENIX NSDI 2013* (2013).

[35] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley DB. In *USENIX Annual, FREENIX Track* (1999), pp. 183–191.

[36] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 29–41.

[37] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Oper. Syst. Rev. 43*, 4 (Jan. 2010), 92–105.

[38] SAGIV, Y. Concurrent operations on B*-trees with overtaking. *J. Comput. Syst. Sci. 33*, 2 (1986), 275–296.

[39] SANFILIPPO, S., AND NOORDHUIS, P. Redis. `http://redis.io`.

[40] SEN, S., AND TARJAN, R. E. Deletion without rebalancing in multiway search trees. *ACM Trans. Database Syst. 39*, 1 (2014), 8:1–8:14.

[41] SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. *ACM Trans. Database Syst. 13*, 1 (Mar. 1988), 53–90.

[42] SHIPMAN, G., WOODALL, T., GRAHAM, R., MACCABE, A., AND BRIDGES, P. InfiniBand scalability in Open MPI. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), pp. 10–pp.

[43] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet: A scalable distributed multiversion B-tree. *Proc. VLDB Endow. 5*, 9 (2012), 884–895.

[44] STUEDI, P., TRIVEDI, A., AND METZLER, B. Wimpy nodes with 10GbE: leveraging one-sided operations in soft-RDMA to boost Memcached. In *Proceedings of USENIX Annual Technical Conference* (2012).

[45] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.

[46] VIENNE, J., CHEN, J., WASI-UR-RAHMAN, M., ISLAM, N., SUBRAMONI, H., AND PANDA, D. Performance analysis and evaluation of InfiniBand FDR and 40GigE RoCE on HPC and cloud computing systems. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on* (2012).

[47] WEI, X., SHI, J., CHEN, Y., CHEN, R., AND CHEN, H. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 87–104.

[48] WINSTON, W. Optimality of the shortest line discipline. *Journal of Applied Probability 14*, 1 (1977).

[49] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion control for large-scale RDMA deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), ACM, pp. 523–536.

An Evolutionary Study of Linux Memory Management for Fun and Profit

Jian Huang, Moinuddin K. Qureshi, Karsten Schwan

Georgia Institute of Technology

Abstract

We present a comprehensive and quantitative study on the development of the Linux memory manager. The study examines 4587 committed patches over the last five years (2009-2015) since Linux version 2.6.32. Insights derived from this study concern the development process of the virtual memory system, including its patch distribution and patterns, and techniques for memory optimizations and semantics. Specifically, we find that the changes to memory manager are highly centralized around the key functionalities, such as memory allocator, page fault handler and memory resource controller. The well-developed memory manager still suffers from increasing number of bugs unexpectedly. And the memory optimizations mainly focus on data structures, memory policies and fast path. To the best of our knowledge, this is the first such study on the virtual memory system.

1 Introduction

The virtual memory system has a long history. It was first proposed and implemented in face of memory scarcity in 1950s [16, 27, 28, 81, 86]. With this technique, the main memory seen by programs can be extended beyond its physical constraints, and the memory can be multiplexed for multiple programs. Over the past several decades, the virtual memory has been developing into a mature and core kernel subsystem, the components and features it has today are far more than the basic functionalities (i.e., page mapping, memory protection and sharing) when it was developed [22].

However, today's virtual memory system still suffers from faults, suboptimal and unpredictable performance, and increasing complexity for development [10, 41, 62, 70, 82]. On the other hand, the in-memory and big memory systems are becoming pervasive today [57, 91], which drives developers to re-examine the design and implementation of the virtual memory system. A quantitative study of the virtual memory system's development process is necessary as developers move forward to next steps. The insights derived from the study can help developers build more reliable and efficient memory management systems and associated debugging tools.

In this paper, we perform a comprehensive study of the open-source Linux memory manager (*mm*). We examine the patches committed over the last five years from 2009 to 2015. The study covers 4587 patches across Linux versions from 2.6.32.1 to 4.0-rc4. We manually label each patch after carefully checking the patch, its descriptions, and follow-up discussions posted by developers. To further understand patch distribution over memory semantics, we build a tool called *MChecker* to identify the changes to the key functions in *mm*. *MChecker* matches the patches with the source code to track the hot functions that have been updated intensively.

We first investigate the overall patterns of the examined patches. We observe that the code base of Linux *mm* has increased by 1.6x over the last five years, and these code changes are mainly caused by bug fixes (33.8%), code maintenance (27.8%), system optimizations (27.4%) and new features (11.0%). More interestingly, we find that 80% of the *mm* patches are committed to 25% of the source code, indicating that its updates are highly concentrated. Such an observation discloses the targeted code regions for our study and future development on virtual memory system.

Furthermore, we examine the bugs in Linux *mm*. We identify five types of bugs: *memory error*, *checking*, *concurrency*, *logic* and *programming*. These bugs are mainly located in the functional components of *memory allocation*, *virtual memory management* and *garbage collection*. Specifically, *mm* is suffering from more concurrency and logic bugs due to its complicated page state management. For example, the memory leaks are mainly caused by the incorrect settings of page states rather than non-freed pages; a significant number of logical incorrectnesses are caused by missing checks on page states.

We further investigate the system optimization patches in *mm*. We identify three major sources: *data structure*, *memory policy* and *fast path*. (1) For *data structure*, we find that 76.2% of patches are committed for software overhead reduction, and 23.8% of them contributed to scalability improvement, across the four popular data structures: *radix tree*, *red-black tree*, *bitmap* and *list*, and their derived structures. (2) For *policy* patches, we find that most of them are concentrated around five design trade-offs: *lazy vs. non-lazy*, *local vs. global*, *sync vs. async*, *latency vs. throughput* and *fairness vs. performance*. For example, OS developers can alleviate overhead caused by expensive operations (e.g., memory com-

Table 1: A brief summary of the Linux *mm* patch study.

	Summary	Insights/Implications
Overview	4 types of patches (i.e., *bug*, *optimization*, *new feature*, *code maintenance*) were committed to 8 major *mm* components (e.g., memory allocation, resource controller and virtual memory management). The patch distribution is highly centralized (§ 3).	(1) the identified 13 hot files from the massive *mm* source code (about 90 files) unveil the focus of the recent *mm* development; (2) with these knowledge, developers can narrow their focus to pinpoint *mm* problems more effectively.
Bug	5 types of bugs (i.e., *checking*, *concurrency*, *logic*, *memory error* and *programming*) have various patterns: *null pointer* and *page alignment* are the popular memory errors; *checking* and *logic* bugs are pervasive due to the complicated page state management (§ 4).	(1) a set of unified and fine-grained page states can be defined to reduce the effort on page tracking for kernel developers; (2) the page state machine should be combined with lock schemes to avoid unnecessary locks; (3) a formal, machine-checked verification framework for *mm* is needed.
Optimization	4 types of *data structure* (i.e., *radix tree*, *red-black tree*, *bitmap* and *list*) optimizations on software overhead reduction and scalability improvement (§ 5.1).	(1) careful examination on nested data structures is necessary to avoid the consequential side effects as we adjust data structures; (2) the internal scalability inside system calls is not well exploited yet.
	Memory *policies* are tackling 5 design trade-offs: *lazy vs. non-lazy*, *local vs. global*, *sync vs. async*, *latency vs. throughput* and *fairness vs. performance* (§ 5.2).	(1) *lazy* policy is preferred as *mm* interacts with fast devices like processor cache, while *async* policy is mostly used for the interaction with slow devices like disk; (2) a large amount of latency-related patches suggest that *mm* profilers are desired to identify more latency-critical operations;
	Fast path has 8 types of approaches: *code reduction*, *lockless optimization*, *new function*, *state caching*, *inline*, *code shifting*, *group execution* and *optimistic barrier*. (§ 5.3).	(1) alleviating redundant functions and reducing lock contentions are the two major contributors for reducing software overhead; (2) these techniques can be generalized and applied in other software systems.
Semantic	35 key functionalities are identified in 13 hot files in Linux *mm*. A majority (75.6%) of them absorb much more patches on *bug fix* and *optimization*. Certain patch pattern is seen for each functionality (§ 6).	(1) the well-developed memory allocators still have tremendous *checking* and *lock* issues due to the increasing complexity of page state management; (2) the fault handler in *mm* is buggy, especially for the cases of out of memory and allocation failures; (3) the patch patterns on memory policy suggest that a policy verification and validation framework is in pressing need;

paction, TLB flush) with lazy policies, but associated checking mechanism has to be implemented to guarantee the program logic is not violated (more patches on this part are committed than the optimization patch itself). (3) We identify eight types of approaches (Table 6) for *fast path* optimization in Linux *mm*, such as *code reduction*, *lockless optimization* and *state caching*.

With the *MChecker* tool, we study the patch distribution over the core *mm* functions to understand the various patterns on memory semantics. Taking the memory policy as example, we categorize it in two types: *policy definition* and *enforcement*. We find that *policy definition* has more issues than *enforcement*. And 30% of the patches were addressing the issues caused by missing checks (e.g., whether page is dirty), missing one check fails the policy enforcement.

We briefly summarize the key findings and present the outline of the paper in Table 1. We discuss the related work in § 7 and conclude the paper in § 8. In the following, we describe the methodologies used in our study.

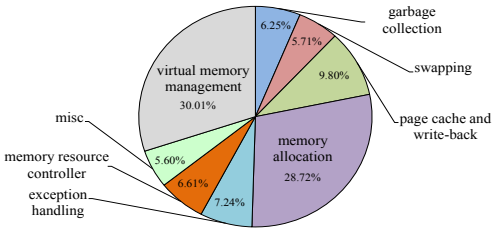


Figure 1: Component breakdown of memory manager in Linux version 4.0-rc4, in terms of lines of codes.

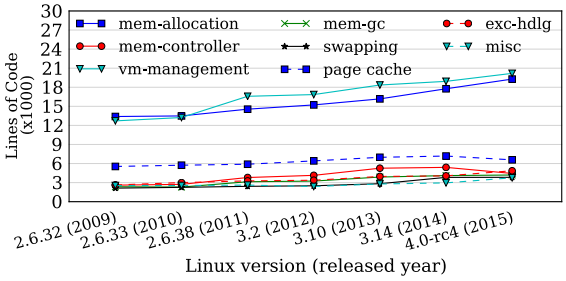


Figure 2: The change in *mm* code in terms of LoC.

2 Methodology

In our study, we target at open-source Linux memory managers, as they provide much more resources (e.g., source code, patches, online discussions) for such a study compared to commercial operating systems. We only select the stable versions that are still supported by the open-source community. The selected Linux kernel versions range from 2.6.32 (released on December 2, 2009) to 4.0-rc4 (released on March 15, 2015), and the time difference between the release dates of two successive versions is 12 months on average. It is noted that Linux 2.6.32 is the oldest version that is still supported. Thus, we believe our study over the past five years represents the latest development trend of the Linux *mm*.

In order to have a comprehensive study of the selected virtual memory system, we manually examine most of the committed patches to Linux memory manager (*root/mm*) following the approaches described in

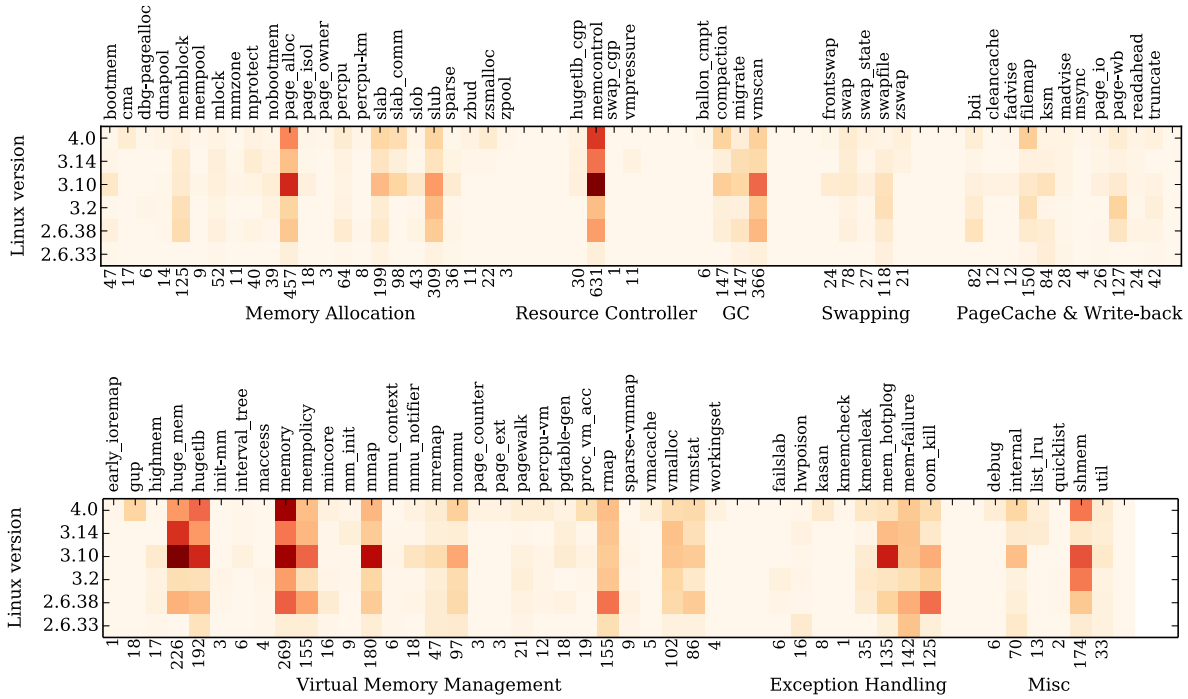


Figure 3: The heat map of patch distribution in each component of Linux memory management. Each block represents the patches committed to the current version since the last stable version. The darker the color, the more patches applied. The number below the bar indicates the total number of committed patches applied to the given file.

[23, 26, 37]. Since December 2, 2009, there are totally 5358 patches relevant to Linux *mm* reported in the patch repositories of Linux kernel. After excluding the duplicated and invalid patches, we examine 4587 patches (85.6% of the total patches). To precisely analyze and categorize each examined patch, we manually tag each patch with appropriate labels after checking the patch, its descriptions, corresponding source code changes, and follow-up discussions posted by developers. The labels include *LinuxVersion*, *CommitTime*, *SrcFile*, *MMComponent*, *PatchType*, *Consequence*, *Keywords*, *Causes*, *Note* and etc. For the patch that belongs to several categories, it will be classified into all the respective categories and studied from different viewpoints. We place all the examined patches into our patch database *MPatch* for patch classification and statistical analysis.

To facilitate our analysis, we break down the Linux *mm* into 8 components according to the functionalities (see Figure 1). We match the examined patches with each component. Taking the Linux version 4.0-rc4 for example, we use the SLOCCount tool [74] to count the line of codes (LoC) in each component. Figure 1 shows the fraction of code serving to accomplish specific functionalities in *mm*. The two largest contributors to Linux *mm* code are memory allocation (28.7%) and virtual memory management (30.0%). This is expected with considering their core functions in virtual memory system. We will discuss how the patches are distributed among these eight components in detail in the following sections.

To further analyze the examined patches, we build a patch analysis tool called *MChecker* to understand the memory semantics by mining the relationships between patches and the key functions in the 'hot' files of Linux *mm*. This will be discussed in detail in § 6.

3 Virtual Memory Evolution

Linux *mm* is constantly updated like other subsystems (e.g., file systems, device drivers) in the Linux kernel. However, few quantitative studies have been done on the Linux *mm*. In our study, we conduct the virtual memory study from the oldest stable version 2.6.32 until the version 4.0, demonstrating what *mm* developers concentrated on over the past five years.

3.1 How is the *mm* code changed?

Taking the Linux 2.6.32 version as the baseline, we examine the source lines of code changes in different Linux *mm* components.We obtain the LoC across different *mm* component in total 7 versions using SLOCCout.

As shown in Figure 2, the LoC is increased in all the *mm* components across successive years compared with the baseline 2.6.32. Overall, Linux *mm* code base has increased by 1.6x over the last five years. *Memory allocation* and *virtual memory management* are the two major components in *mm*, the updates to the two components constantly occupy a large portion of the overall patches.

Understanding the code changes is important for us to pinpoint how the Linux *mm* is evolved. More detailed analysis is given on where and why the *mm* code has been changing in the following.

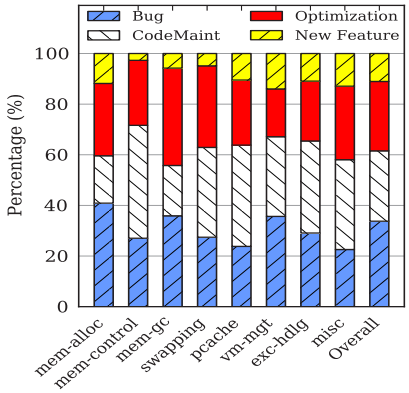


Figure 4: Patch overview. It shows the patch distribution according to general types including *bug*, *code maintenance*, *improvement* and *new feature*.

3.2 Where is the *mm* code changing?

The patches applied to Linux *mm* record all the changes to its code base and provide the evidences showing how one version transforms to the next. Figure 3 demonstrates the patch distribution among all the components in Linux *mm*. One patch may be applied to several files in *mm*, we count it to all the involved files. The average source LoC changed in each patch is 62, it is much less than the source LoC in feature patches. For example, the compressed swap caching (*zswap*) was introduced in 2013 [84], and a new file named `zswap.c` with 943 LoC was added in the code base of Linux *mm*.

We identify several interesting findings via the heat map. First, the patches are concentrated around only a few files in each component (see the darker column and blocks in the heat map of Figure 3). About 80% of the patches were applied to the 25% of the source code. These hot files generally represent the core functions of the corresponding component. Second, the patches applied to these hot files are much more than other files. For instance, the number of patches relevant to *huge_mem* in *virtual memory management* component is about 12x more than that of 'cold' files. Third, for these hot files, most of them are constantly updated along the Linux evolution from one version to the next. Typical examples include the *memcontrol* in memory resource controller, the *memory* in virtual memory management.

It is understandable that more patches are committed between Linux 3.2 and 3.10 compared to other intervals, as the time between the two versions is 19 months which is longer than the average time interval (12 months).

3.3 Why is the *mm* code changed?

We identify that the *mm* source code changes come from four sources: *new feature*, *bug fixes*, *optimization* and *code maintenance*. We classify the patches into these four categories, and examine how each category contributes to the evolution of Linux *mm*.

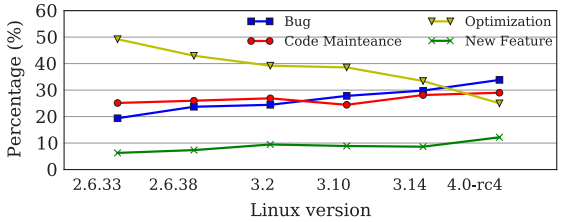


Figure 5: The changes of patch distribution along the Linux *mm* evolution, taking Linux 2.6.32 as the baseline.

Figure 4 shows the patch distributions among the 8 components. Overall, 33.8% of the patches are applied for bug fixes, 27.8% of the patches are relevant to code maintenance, 27.4% are for system optimizations, and 11.0% are new features. Common sense suggests that virtual memory system has been developed into a mature system, our findings reveal that the bug fixes are still the main thread of patch contributions.

Furthermore, we examine how the four types of patches changed over time. As shown in Figure 5, we find that *bug* patches are increasing steadily, indicating more bugs are expected in Linux *mm* as the complexity of its code base is increasing (see Figure 2). The percentage of *code maintenance* and *new feature* patches keep at a constant level in general, but a slightly increase in *new feature* patches is seen recently. Perhaps most interestingly, *optimization* patches are decreasing over time, which can be expected as Linux *mm* becomes more adapted to current systems (e.g., multi-core processors).

Summary: Linux *mm* is being actively updated, The code changes are highly concentrated around its key functionalities: 80% of the patches were committed to the 25% of the source code.

4 Memory System Bugs

In this section, we examine the *bug* patches in detail to understand their patterns and consequences.

4.1 What are *mm* bugs?

With the tagging of these patches, we classify the *bug* patches into 5 general types: *memory error (MErr)*, *checking*, *concurrency*, *logic* and *programming*. Each general type is further broken down into multiple subtypes according to their causes, as shown in Table 2. Like the systems such as *file systems* [37] and *device drivers* [29], many bugs are general software bugs (e.g., *programming* bugs). In this paper, we are more interested in memory-related bugs, for example the *alignment* bugs in *MErr*, and the *logic* bugs (§ 4.2).

4.2 How *mm* bugs are distributed?

The heat map of Linux *mm* bug distribution among its eight components is shown in Figure 6. More bugs lie in

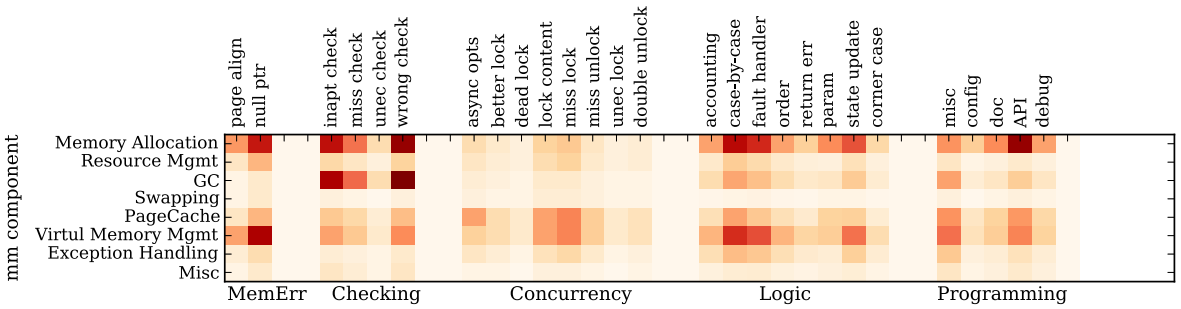


Figure 6: Bug distribution among Linux *mm* components.

Table 2: Classification of *bug* patches.

	Sub-type	Description
MErr	alignment	data/page alignment and padding.
	null pointer	refer to an invalid object.
Checking	inapt check	inappropriate check.
	miss check	check is required.
	unec check	unnecessary check.
	wrong check	check conditions are incorrect.
Concurrency	async opts	faults due to async operations
	better lock	lock is implemented in better way.
	dead lock	two or more threads are blocked.
	lock contention	concurrently access to shared locks or data structures.
	miss lock	lock is required.
	miss unlock	the unlock is missed.
	unec lock	unnecessary lock.
	double unlock	unlock twice.
Logic	accounting	error in collecting statistics.
	case-by-case	bug fix requires specific knowledge.
	fault handler	error and exception handling.
	order	the order of execution is violated.
	return err	return code is not correct.
	parameter	misuse of parameters.
	state update	issues in updating state and data structures.
	corner case	uncovered cases in implementations.
Programming	configuration	wrong/missed configuration.
	document	comments & docs for functions.
	API	issues caused by interface changes.
	debug	issues happened in debugging.
	misc	any other programming issues

the three major components *memory allocation*, *virtual memory management*, and *GC*, which matches with the patch distribution as shown in Figure 3. More specifically, we identify several interesting findings in *mm*:

Memory Error (MErr): We find that *null pointer* dereferences (e.g., [45, 67, 76]) are the most common bugs because of the missing validations of pointers before using them in *mm*. These bugs happened even in *mm*'s core functions such as slub, which is unexpected. The alignment of data structures and pages are important factors in *mm* optimizations, however bugs frequently happen at boundary checking and calculations for padding (e.g., [5, 6]). As they usually involve many shift operations, validating the correctness of the bit manipulations is necessary.

Checking: As *mm* involves many state checking operations, especially in *memory allocation* and *garbage collection* (GC) components. The *checking* bugs appear frequently due to inappropriate and incorrect checking, for instance, the GC has to check if a page is used or not before the page migration is issued; *free_bootmem_core* may free wrong pages from other nodes in NUMA without correct boundary checking [47].

Concurrency: We find that more *miss lock* and *lock contention* bugs appeared in *virtual memory management* due to the complicated page states, and more efforts are required for kernel developers to track the page states. In addition, the page state machine can be combined with lock schemes to avoid unnecessary locks, for instance, when kernel pages are charged or uncharged, the *page_cgroup* lock is unnecessary as the procedure has been serialized (e.g., [40]).

Logic: We identify three important *logic* bugs: *case-by-case*, *state update* and *fault handler*. For the first two types, they may not stall the system or generate exceptions immediately, but they make the system execute in unexpected workflow or states, resulting in incorrect states or *runtime error* eventually. Fixing these bugs often require domain specific knowledge. For example, when *shmem* intends to replace a swap cache page, the original implementation calls *cgroup* migration without *lrucare* based on the incorrect assumption that the page is not on the LRU list. As for *fault handler* bugs, many of them were caused by lack of or inappropriate implementation of exception handling (e.g., [56, 77, 85]).

There is still a long way to have a bug-free virtual memory system. It is much hard for formal proof to verify the correctness of *concurrency* events [31], and few previous work has the formal, machine-checked verification for virtual memory system specifically.

4.3 What are the *mm* bug consequences?

We further examine the consequences of *mm* bugs to understand how serious they are. We classify the bug consequences into 7 types with reference to the classification in [37]. Figure 7 shows that *logic* bugs lead to wrong behaviors and runtime errors with higher chances. *Concurrency* bugs are more likely to make system crash or hang, since they often produce *null pointers* and *deadlocks* if

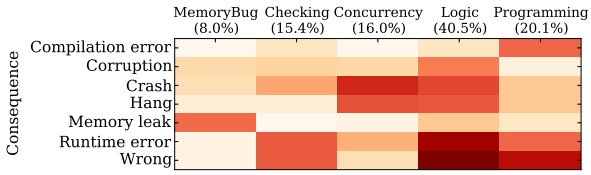


Figure 7: Bug consequence.

the shared data structures are not maintained properly.

Moreover, we find that *checking* bugs are also the major contributors to the wrong behaviors and runtime errors in Linux *mm*. More interestingly, we find that the memory leaks are mainly caused by *MErr* and *logic* bugs. It is noted that most of the memory leaks in *mm* are not caused by not-freed memory, they are mostly caused by the *accounting* (e.g., the unused page is not counted as free page) and *fault handler* bugs (e.g., pages are not reclaimed when fault happens). For *programming* bugs that mainly cause compilation errors, runtime error and wrong behaviors, they are easier to be fixed compared with other types of bugs.

Summary: Memory leaks in *mm* were mainly caused by the *accounting* bugs and inappropriate implementation of *fault handler* (e.g., page fault), instead of non-freed memory. The complex states of pages complicate the implementation of the *checking* and *locking* mechanism, which requires large effort for kernel developers to track the correct states.

5 Memory Optimizations

As discussed in § 3, *optimization* (27.4% of total patches) is one of the major contributors to the code changes in Linux *mm*. We identify several sources that contributed to the optimizations in *mm*: *data structure*, *policy* and *fast path*. In this section, we will explain how these optimizations were performed in improving Linux *mm*.

5.1 Memory Data Structures

In virtual memory system, data structure is one the critical factors for its efficiency [10, 15, 20]. Likewise, the data structures in Linux *mm* are constantly tuned to reduce software overheads, and specialized data structures are leveraged for special purposes such as page lookup and memory allocation.

5.1.1 What are the common data structures?

We identify four popular data structures in Linux *mm*: *radix tree*, *red-black tree*, *bitmap* and *list*, according to their relevant patch distributions.

Radix tree [65, 66] is typically used within *address_space* structure in Linux *mm* for tracking in-core pages for page caches, because of its storage efficiency for sparse trees. **Red-black tree** [69] such as the one in *vm_area_struct* can perform lookups in logarithmic

```
1: struct memcg_cache_params {
2:   bool is_root_cache;
3:   union {
4:     struct kmem_cache  *memcg_caches[0];
5:     struct {
6:       struct mem_cgroup *memcg;
7:       struct list_head list;
8:       struct kmem_cache *root_cache;
9:       bool dead;
10:      atomic_t nr_pages;
11:      struct work_struct destroy;
12:    };
13:  };
14: };
```

Linux 3.8

```
struct memcg_cache_params {
  bool is_root_cache;
  struct list_head list;
  union {
    struct memcg_cache_array
            __rcu *memcg_caches;
    struct {
      struct mem_cgroup *memcg;
      struct kmem_cache *root_cache;
    };
  };
};
```

Linux 4.0

Figure 8: Comparison of *memcg_cache_params* structure in Linux version 3.8 and 4.0.

Table 3: Typical examples of approaches to reduce software overhead of different data structures.

Type	Overhead Source	Optimization Example
radix tree	Tree walking	Provide hints, cache intermediate states [48]
	Linear search	Add bit-optimized iterator [64]
rb tree	Tree walking	Optimized tree walking [1, 44]
	Lock contention	Batch lookup [11]
	Balancing tree	Reduce lazy operations [52]
list	List search	Limit list length [43]
	Lock contention	Add per-node LRU list [35]
	Storage overhead	Dynamic allocation

time, and its insert and delete operation can be finished in bounded time. It is used to track *VMA*s. **Bitmap** is usually used to index pages in RAM, which involves bit manipulation frequently. Besides these specific data structures, other data structures such as **list** are widely used in Linux *mm*. Most of the recent patches are related to their derived data structures, such as LRU list which is used by multiple *mm* components to track page access frequency.

5.1.2 How are data structures optimized?

We identify that the optimization of *mm* data structures mostly focuses on two aspects: software overhead reduction (76.2%) and scalability improvement (23.8%).

Reducing software overhead. In Linux *mm*, we find that the software overhead on these data structures mainly come from the following sources: *tree walk*, *tree balance*, *lock contention* and *storage cost*. A variety of approaches have been applied to address these issues as shown in Table 3. For instance, to reduce lock contentions, multiple lookups can be performed in batch once a lock is acquired.

Reorganizing data structures is another approach that usually adopted to improve memory efficiency. However, this approach may introduce additional overhead that offsets its benefits. Beyond our expectation, a significant portion of patches were applied to avoid the extra overhead caused by nested data structures. An interesting example is shown in Figure 8. The structure *memcg_cache_param* in Linux version 4.0 shrinks compared to its initial design in version 3.8. However, the saved memory does not justify, as the pointer dereference

in its correlated structure *kmem_cache* may cause extra cache line access. Thus, the pointer was replaced with embedded variable [73]. As we adjust data structures in virtual memory system, their referenced data structures are often overlooked, producing side effects.

Improving scalability for data structures. Scalability issue is another major aspect of data structure optimizations. Most of the scalability issues are caused by locking for atomic access to shared data structures. We find that a main thread of the applied patches is to decentralize the data structures and replace the shared counterparts with per-core, per-node and per-device approaches.

With the increasing memory size and core counts, the scalability issues become more prominent [10, 90], appropriate usage of data structures is critical for both performance and memory efficiency, such as recent work [11] suggested that replacing red-black tree with radix tree to track non-overlapping virtual memory regions for *mmap* and *munmap* provides better scalability. Furthermore, the system support for new memory technologies like persistent memory bring new scalability challenges, the dramatically increased physical memory capacity generates large pressure on memory management, e.g., a 1 TB of persistent memory with 4 KB page size requires 256 million page structures [14, 61].

To scale OS kernels, Clements et al. [12] proposed a commutativity tool to guide the implementation of high-level APIs, however it cannot expose the internal scalability issues (e.g., global vs. local data structure) inside the system calls. Our findings on *mm* data structures suggest that it is necessary to build tools to check the bottlenecks introduced by global and shared data structures.

Summary: The software overhead and scalability issues caused by data structures remain big concerns for OS developers: more efforts on system-wide optimization for nested data structures, and the internal scalability inside system calls are required.

5.2 Policy Design: Tackling Trade-offs

Memory is one of the most desired yet constrained resource in computer systems, multiple design trade-offs have to be made to fully utilize the resource and to improve performance. We find that a majority of the patches relevant to the policy design are concentrated on tackling these trade-offs. Through the patch study, we expect to learn from the lessons with policy designs and implementations conducted by OS developers.

5.2.1 What are the trade-offs?

Based on our patch study, we summarize the trade-offs that OS designers have frequently tackled in Table 4, and also present a case study for each of them. The software overhead caused by expensive operations, such as memory compaction, page migration and TLB flush, can be significantly alleviated using asynchronous or lazy policies. However, such benefit is not free because they complicate the program logic, leading into serious runtime errors like data inconsistency. We will present how the policy design decisions were made in Linux *mm*, with a focus on the new class of *mm* optimizations.

Table 4: Classification of typical design choices in Linux *mm* based on the analysis of *optimization* patches.

Trade-off	%	Case Study
Latency Vs. Throughput	10.9	disk access upon page fault and swapping.
Synchronous Vs. Asynchronous	22.3	With asynchronous method, *mm* can avoid delays while executing expensive operations like swapping, compaction.
Lazy Vs. Non-lazy	15.6	Expensive operations (e.g., TLB flush, page migration) can be executed in batch.
Local Vs. Global	33.1	Maintaining per-process variables improves scalability, but it increases storage overhead, e.g., slub vs. slab allocator.
Fairness Vs. Performance	18.1	Fairness guarantee when memory is shared among multiple processes.

Table 5: Examples of applying lazy policy in Linux *mm*.

Functionality	Example
vmalloc	lazy TLB flush, lazy unmapping
mempolicy	lazy page migration between nodes
huge_memory	lazy huge zero page allocation
frontswap	lazy backend initialization
cleancache	lazy backend registration
backing-dev	lazy inode update on disk

5.2.2 How are the policy decisions made in *mm*?

Latency matters in memory manager. This trade-off of latency vs. throughput centers around the *page cache and write-back* component in *mm*. The I/O requests are issued in batch and served in a disk-friendly order to exploit the full bandwidth of disks for high throughput, but it may increase I/O latency. For instance, the original design of *readahead* component favors sequential access for higher I/O throughput, making the average latency of random reads disproportionately penalized [68]. A general approach on decision-making for this trade-off is to prioritize the dominate workloads patterns, so systems pay little or acceptable cost on its downsides.

More interestingly, we identify 137 patches committed specially for reducing the latencies of *mm* operations (e.g., page allocation, vma lookup, page table scanning). As in-memory systems are becoming pervasive and latency matters to today's application services, it is worthwhile to build *mm* profilers or test framework to identify more latency-critical operations.

Async is popular, but be careful to its faults. Asynchronous operations are widely used to hide expensive operations in *mm*. For example, async compaction was

Table 6: Classification of approaches for fast path optimization in Linux *mm*.

Type	%	Description	Example
Code Reduction	34.1	Simplify the fast path logic and reduce redundant codes.	Avoid redundant `get/put_page` in `munlock_vma_range` as pages will not be referred anymore [50].
Lockless Optimization	27.3	Reduce the usage of lock and atomic operations.	Lockless memory allocator in SLUB [34, 36].
New Function	11.6	Improve with new *mm* functionality.	New SLUB fast paths are implemented for `slab_alloc/free` with `cmpxchg_local` [75].
State Caching	8.2	Cache the states to avoid expensive functions.	Pre-calculate the number of online nodes instead of always calling expensive `num_online_nodes` [59].
Inline	6.4	Inline simple functions in fast path.	Inline `buffered_rmqueue` [58].
Code Shifting	4.7	Move unfrequently executed code from fast path to slow path.	In SLUB allocator, slow path executes the irq enable/disable handlers, fast path will execute them only at fallback [78].
Group Execution	4.1	Avoid calling the same function repeatedly.	Using `pte_walk` to avoid the repeated full page table translation and locks in `munlock_vma_pages_range` [49].
Optimistic Barrier	3.6	Optimistically skip or reduce the number of barriers, and re-execute the logic once false failure is detected.	Using only read barriers in `get/put_mems_allowed` to accelerate page allocation [13].

introduced to reduce the overhead caused by expensive memory compaction; the expensive *trim* command [80] in SSDs should be issued in parallel with other I/O operations before actual writes happened due to its long latency. We find that the common issues in implementing async mechanisms located in their fault handlers for exceptions (e.g., early termination [46]).

Trying lazy policy to alleviate expensive operations. The key idea of lazy policy is to delay several expensive operations, and batch them into a single operation or system call if semantics are allowed. Table 5 shows a set of cases that have leveraged lazy policy to reduce the frequency of expensive operations. In contrast to *async* policy used usually as *mm* interacts with slow devices, *lazy* policy is more beneficial when *mm* components interact with fast devices (e.g., CPU, processor cache, TLB) according to our patch studies.

Since lazy policy may change the execution order of subsequent functions, which would make systems in unexpected states temporarily, careful examination should be conducted as we decide whether a specific function should be delayed or not. For instance, the operation of flushing virtual cache on *vunmap* in *pcpu_unmap* cannot be deferred as the corresponding page will be returned to page allocator, while TLB flushing can be delayed as the corresponding *vmalloc* function can handle it lazily [18].

Decentralizing global structures for better scalability. As seen in our patch study, more per-node, per-cpu variables are replacing their global counterparts to improve the system scalability. For example, new dynamic per-cpu allocator was introduced to avoid the lock contention involved in memory allocations. This approach has also been widely adopted in other subsystems such as device drivers, CPU scheduler and file systems.

Memory resource scheduling for fairness and performance. The trade-off between fairness and performance is a well-known yet hard problem. In Linux *mm*, we find that this type of patches mainly concentrated on the *memory allocation and reclamation*. In page allocation, round-robin algorithm is used to guarantee *zone fairness*. However, this algorithm did not consider the latency disparity across zones, resulting in remote memory reference and performance regression. During page reclamation, the allocator reclaimed page in LRU order which can only provide the fairness for low order pages but not for pages at higher order, which could penalize the performance of the applications (e.g., network adapter) that desire high-order pages [38].

Summary: Most of the *policy* patches are tackling five types of trade-offs. The experiences with *mm* development provide us the hints on how and where each policy would be leveraged in practice.

5.3 Fast Path

To further reduce the software overhead, another optimization approach is to accelerate the commonly executed codes, which is named as *fast path*. We identify that Linux *mm* maintains fast paths in most of its key components, such as *memory allocation*, *memory resource controller*, and *virtual memory management*. These fast paths are carefully and frequently re-examined in every version of Linux kernels, contributing many patches to the code base of Linux *mm*. We study these patches to understand what are the common techniques leveraged in fast path optimizations.

Based on our patch study, we categorize the techniques used for fast path in Linux *mm* into eight types as described in Table 6. We find that *code reduction* and *lockless optimization* are the most commonly used techniques for fast path, which contributed 61.4% of the fast path related patches. As case studies, we list a set of patches for these types in Table 6. For instance, on the fast path for the allocation and deallocation of page table pages, there are two costly operations: finding a zeroed page and maintaining states of a slab buffer. To reduce these overheads, new component called *quicklist* was in-

troduced to replace the allocation logic as it touches less cache lines and has less overhead of slab management. Another interesting approach is *optimistic barrier*, which is proposed to reduce the synchronous overheads caused by system call like *barrier* and *fence*, e.g., the full memory barriers on both reads and writes are replaced with only read barriers on the fast path, while re-executing the code with slow path when false failure is detected.

Summary: There are 8 types of optimizations for fast path. *Code reduction* and *lockless optimization* are the two most widely used techniques to alleviate redundant functions and reduce lock contention.

6 Memory Semantics

In order to better understand memory semantics, we build a tool named *MChecker* to pinpoint the modified functions in source code by leveraging the information (e.g., which lines of code are changed) provided in patches. *MChecker* will place the located functions under the record of the corresponding patch in *MPatch*. By analyzing the call graphs of specific memory operations, we can identify their core functions. And with the information provided by *MPatch*, we can easily understand what are the common bugs lying in these functions, how these functions were optimized and etc.

In this paper, we analyze the hot files which are evolved with more committed patches (see Figure 3). Due to the space limitation of the paper, we only present the hot functions that were updated intensively in Table 7. Across the 35 major functionalities (the 3rd column in Table 7), 75.6% of them have more patches for *bugs* and *optimization* than those for *code maintenance* and *new feature*, which demonstrates the main thread of contributions from the open-source community.

6.1 Memory Allocation

The memory allocation and deallocation functions in the kernel space are mostly implemented in `page_alloc`, `slab`, `slub` and `slob` files which are the cores of the well-known buddy system. As the default memory allocator, `slub` absorbs 1.6-7.2x more patches than other two allocators. The functions in these hot files can be categorized into *allocation/free*, *create/destroy* and *page/cache management* to fulfill the implementation of the user-space `malloc` and `free` functions. As memory allocators become mature, about half of the relevant patches are concentrated on the page and cache management (e.g., cache initialization within `kmem_cache_init`, object state maintenance within `show_slab_objects`).

We find that the mature memory allocation and deallocation are still suffering from bugs. The *allocation* and *create* functions have more bugs than *free* and *destroy* functions, and these bugs are usually relevant to *checking* and *lock contentions*. The cumbersome checks and lock protections (due to complicated page states) not only increase the possibility of bug occurrence, but also incur increasing software overhead. To reduce such software overhead, an increasing number of improved versions of allocation functions with *fast path* are implemented. For instance, a lockless allocation algorithm based on the atomic operation `this_cpu_compxchg_double` improves the allocation performance significantly [83].

Summary: The mature memory allocators still suffer from serious *checking* and *lock* issues due to the complicated states maintained for memory pages during their life cycles.

6.2 Memory Resource Controller

In memory resource controller, the majority (93.8%) of its patches are committed to the `memcontrol` file. As more resource is available on today's machines, the OS-level resource control named *memory cgroup* was proposed to support resource management and isolation. It is motivated to isolate the memory behavior of a group of tasks from the rest of the system.

To control the usage of memory, *charge/uncharge* functions are used to account the number of pages involved along with the running tasks. We find that 26.2% of the committed patches relate to *concurrency* issues, because of the complicated intrinsic operations in these functionalities. For instance, for a uncharging page, it may involve actions of truncation, reclaim, swapout and migration. Interestingly, many of these issues are caused by missing locks. However, detecting missing locks is more than a software engineering issue, as it requires program semantics (e.g., page states) provided by the virtual memory system. Such an observation may give us the hint that the decoupled memory resource controller should be integrated into the *mm* framework to avoid redundant data structures and software overheads, e.g., *memcontrol* can rely on the existing LRU lists to obtain page information and schedule pages dynamically.

Moreover, we find that fault handlers suffer from a significant portion of bugs, the involved cases include out of memory (OOM) and allocation failure. Similar trends are seen in the *exception handler* component which has two hot files: `memory-failure` and `oom_kill`. Most of them are caused by the inappropriate or wrong handling of memory errors and exceptions (e.g., [8, 19]). We expect these findings would reveal the weakness aspect of Linux *mm* and supply useful test cases to the memory testing and debugging tools like mmtests [42, 53].

Summary: *Concurrency* issues (e.g., missing lock) are the major concern for the memory resource controller development. Our study discloses that the fault handler is still a weak aspect in *mm*.

Table 7: The heat map of patches for the hot functions in *mm*. Functions in each hot file are categorized into sub-types (the 3rd column). The proportion of each functionality is illustrated in the 4th column. The distribution of *bugs* (BUG), *code maintenance* (CODE), *new feature* (FTR) and *optimization* (OPT) is shown across 5-8th columns.

	File	Functionality	%	BUG	CODE	FTR	OPT	Representative Hot Functions
Memory Allocation	page_alloc	Allocation	24.6	8.1	6.8	2.2	7.4	alloc_pages, alloc_pages_slowpath, _rmqueue
		Free	19.4	4.8	8.3	1.3	5.0	free_one_page, free_pages_bulk, free_zone_pagesets
		Page management	56.0	14.1	22.0	4.4	15.5	build_zonelists, zone_init_free_lists, read_page_state
	slab	Create/destroy	11.6	4.1	4.9	1.5	1.2	kmem_cache_create, kmem_cache_destroy
		Allocation/free	29.8	11.6	12.4	2.0	3.8	cache_alloc, kmalloc, kmem_cache_free
		Shrink/grow	6.9	1.7	3.2	0.6	1.4	cache_grow, cache_reap
		Cache management	51.7	8.7	22.2	1.8	19.1	kmem_cache_init, kmem_getpages
	slub	Create/destroy	14.9	5.4	4.5	0.4	4.5	kmem_cache_create, kmem_cache_destroy
		Alloction/free	33.8	9.5	8.1	0.9	15.3	slab_alloc, kmalloc_large_node, kfree, slab_free
		Slub management	51.4	22.1	9.5	4.1	15.8	kmem_cache_init, show_slab_objects
Controller	mmcntr	Charge/uncharge	26.5	9.3	1.0	2.4	13.8	mem_cgroup_do_charge, mem_cgroup_try_charge, mem_kmem_commit_charge
		Cgroup management	73.5	35.4	7.9	7.2	23.0	mem_cgroup_alloc, mem_cgroup_handle_oom
Expt Hndler	failure	Fault handler	75.9	38.9	13.0	9.3	14.8	memory_failure, collect_procs_file
		Hwpoison	24.1	13.0	1.9	3.7	5.6	hwpoison_user_mappings, hwpoison_filter_task
	OOM	Candidate task	53.7	14.8	16.6	7.4	14.8	oom_badness, select_bad_process, oom_unkillable_task
		OOM handler	46.3	24.1	5.6	3.7	13.0	out_of_memory, oom_kill_process
Virtual Memory Management	memory	Page table	26.3	11.6	10.0	1.6	3.1	do_set_pte, pte_alloc, copy_one_pte, zap_pte_range
		Page fault	23.4	8.8	7.3	1.5	5.9	do_fault, handle_mm_fault, do_shared_fault
		Paging	25.5	7.3	9.5	5.1	3.6	vm_normal_page, do_anonymous_page, do_swap_page
		NUMA support	18.2	6.3	2.1	1.4	8.4	do_numa_page, access_remote_vm, numa_migrate_prep
		TLB	6.6	2.2	0.5	1.1	2.8	tlb_flush_mmu, tlb_gather_mmu
	mpol	Policy definition	47.7	17.5	15.1	4.7	10.5	mpol_new, mpol_dup, mpol_shared_policy_lookup
		Policy enforcement	52.3	17.4	19.8	9.3	5.8	do_mbind, do_set_mempolicy, vma_replace_policy
	hugemm	Page table support for hugepage	68.5	33.7	19.1	2.3	13.5	change_huge_pmd, do_huge_pmd_numa_page, copy_huge_pmd
		Hugepage alloc	31.5	9.0	13.5	1.1	7.9	hugepage_init, alloc_hugepage, khugepaged_alloc_page
	hugetlb	Hugepage management	33.7	20.5	4.8	2.4	6.0	alloc_huge_page, free_huge_page, hugetlb_cow
		Hugepage fault	8.4	3.6	1.2	2.4	1.2	hugetlb_fault
		VM for hugepage	57.8	19.2	27.7	7.2	3.6	hugetlb_change_protection, vma_has_reserves
	mmap	Mmap operations	31.7	13.3	10.0	5.0	3.3	do_mmap_pgoff, do_munmap, exit_mmap
		VM for mmap	68.3	23.3	21.7	8.3	15.0	mmap_pgoff, mmap_region, vma_adjust
	alloc	Vmalloc	48.9	17.8	22.2	2.2	6.7	vmalloc_node_range, vmalloc_area_node, vmalloc_open
		Vmap	51.1	13.3	26.7	2.2	8.9	alloc_vmap_area, vunmap, free_vmap_area
GC	vmscan	Kswapd	20.0	5.5	7.3	1.8	5.5	kswapd, wakeup_kswapd
		Shrinker	52.7	12.7	16.3	7.3	16.3	shrink_inactive_list, shrink_page_list, shrink_zones
		GC helper	27.3	7.3	3.6	1.8	14.6	get_scan_count, pageout, scan_control

6.3 Virtual Memory Management

The virtual memory management is the largest component in Linux *mm*, it has six hot files: `memory`, `memorypolicy`, `huge_memory`, `hugetlb`, `mmap` and `vmalloc`. These hot files which contain the essential elements in virtual memory (e.g., page table, page fault handler, paging and TLB) has the largest number of committed patches (see Figure 3). These essential functions which have been developed for decades are still being updated to support new hardwares (e.g., NVDIMM and persistent memory [61]), and new usage of memory (e.g., huge page). And they are still buggy, e.g., even in the well-developed `do_set_pte` function, missing setting the *soft dirty* bit [79] could cause data inconsistency if a user space program is tracking memory changes [17].

A core component of memory management is the design and implementation of memory policies. We generally categorize the functions for memory policies into two categories: *policy definition* and *policy enforcement*. We find that *policy definition* has more *optimization* patches than *policy enforcement*, for instance, new policy is defined for choosing preferred NUMA node based on the number of private page faults; the kernel should avoid immediate memory migrations after switching nodes. As for *bugs* in memory policies, we find that 30% of the patches were applied to address the issues caused by

missing checks, since memory policies usually count on many states (e.g., whether page is dirty) and statistics (e.g., cache hit/miss rate, number of page fault), missing one check would fail the policy enforcement.

Summary: The well-developed virtual memory management is still buggy. Specifically, a large portion of issues in *policy definition* and *enforcement* are caused by missing checks.

6.4 Garbage Collection

To preserve a certain amount of available memory pages for future allocation, Linux *mm* will garbage collect unused pages across memory zones. In `vmscan`, its functions can be categorized into two major categories: *kswapd* (the kernel swap daemon) and *shrinker*. For the remaining functions, we define them as *GC helper*, as they either provide the statistics of page usage, or handle the page out for *shrinker*. The *kswapd* will try to free pages if the number of free pages in the system runs low, while *shrinker*-relevant functions will be called to scan the corresponding components (e.g., slab cache, zones), and locate the candidate pages to be freed. Most interestingly, we find that more patches (52.7%) were applied to *shrinker* functions, and 84.1% of them relate to memory policies, focusing on how to scan memory regions with low overhead, and which pages should be reclaimed. However, the congestion events (22 related patches, e.g., [39, 51, 87]) during garbage collection make the memory performance unpredictable, it is essential to scale the GC process (e.g., per-zone scanning) and use page statistics to coordinate GC activities.

Summary: The *shrinker* is the hot spot in GC, it causes unpredictability in memory performance. A scalable and coordinated GC is desirable.

7 Related Work

The key components of virtual memory were studied back to 70's when DRAM first appeared [16]. Over the past decades, the functionalities of the core components have been enriched markedly, such as the buddy system [33] was proposed in 90's to remove the internal fragmentation, and later, general OS support for huge pages was proposed [21]. Today, the virtual memory system is still actively improved to support new features [3, 30, 72] and hardware [25, 32, 54, 71, 88]. There is no doubt that the memory manager has become one of the core subsystems in today's OS kernel. But few work has done any studies on the development of virtual memory system recently (over the past decade). Gorman et al. [22] analyzed the *mm* source code in Linux version 2.6.0-test4 (2003), while our work focuses on the study of patches which were committed to the latest Linux versions (from 2.6.32, 2009 to 4.0-rc4, 2015).

Patch and bug studies provide us insights on issue patterns in specific system software, and the experiences along its development. A number of such studies in various systems have been conducted recently. Lu et al. [37] studied the Linux file system patches, Chou et al. [9] investigated the operating system errors in Linux kernels, Kadav et al. [29] examined the code base of Linux device drivers, Palix et al. [60] studied the Linux kernel to version 2.6.33. We share the same purposes with these studies, but with a focus on Linux virtual memory system. To the best of our knowledge, our work is the first to conduct such a comprehensive study, which examines 4587 committed patches to Linux memory manager.

Memory issues can lead to serious problems, such as system crash, data corruption, suboptimal performance, etc. Many tools [2, 4, 7, 55] were built to address memory-related errors such as buffer overflows, dangling pointers, memory leaks in the programming and library space. Few of the related work is targeting at addressing the bugs in kernel space. Yang et al. [89] built checkers to find the bugs in storage and file systems, Prabhakaran et al. [63] developed a file system with better failure policies. Such efforts are also required as we build more reliable and efficient virtual memory systems. And also, a specific formal verification [24, 31] for virtual memory system is needed to verify its policies and implementation. The insights and bug patterns disclosed in this work would facilitate the development of these tools.

8 Conclusion

In this paper, we present a comprehensive study of 4587 committed patches in the Linux memory manager over the last five years. These patches reflect the development of the virtual memory system. We expect our findings to benefit the development of existing and future virtual memory systems, and their associated bug-finding and debugging tools. Our study would also shed light on the development of memory manager in relevant OSes as they share the same principles as Linux kernel.

Acknowledgments

This paper is dedicated to the memory of our colleague and mentor Karsten Schwan. Karsten was a prolific researcher, a great advisor, and a faculty leader in the School of Computer Science at Georgia Institute of Technology. The lead author greatly thanks him for his mentorship and guidance. We all miss you Karsten.

We thank our shepherd Zhen Xiao as well as the anonymous reviewers. We also thank Xuechen Zhang for the initial help and discussions on this work. This work was supported in part by the Center for Future Architectures Research (CFAR), one of the six SRC STARnet Centers, sponsored by MARCO and DARPA.

References

[1] ADD RBTREE POSTORDER ITERATION FUNCTIONS, RUNTIME TESTS, AND UPDATE ZSWAP TO USE. https://lwn.net/Articles/561124/.

[2] AKRITIDIS, P. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security'10* (2010).

[3] BASU, A., GANDHI, J., CHANG, J., HILL, M. D., AND SWIFT, M. M. Efficient Virtual Memory for Big Memory Servers. In *ISCA'13* (Tel-Aviv, Israel, June 2013).

[4] BERGER, E. D., AND ZORN, B. G. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI'06* (Ottawa, Ontario, Canada, June 2006).

[5] BOOTMEM: FIX CHECKING THE BITMAP WHEN FINALLY FREEING BOOTMEM. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/bootmem.c?id=6dccdcbe2c3ebe152847ac8507e7bded4e3f4546.

[6] BOOTMEM: FIX FREE_ALL_BOOTMEM_CORE() WITH ODD BITMAP ALIGNMENT. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/bootmem.c?id=10d73e655cef6e86ea8589dca3df4e495e4900b0.

[7] BRUENING, D., AND ZHAO, Q. Practical memory checking with Dr. Memory. In *CGO'11* (Apr. 2011).

[8] CALL SHAKE_PAGE() WHEN ERROR HITS THP TAIL PAGE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memory-failure.c?id=09789e5de18e4e442870b2d700831f5cb802eb05.

[9] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. An Empirical Study of Operating Systems Errors. In *SOSP'01* (Dec. 2001).

[10] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. Scalable Address Spaces Using RCU Balanced Trees. In *ASPLOS'12* (London, England, UK, Mar. 2012).

[11] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Eurosys'13* (Prague, Czech Republic, 2013).

[12] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. In *SOSP'13* (Farmington, Pennsylvania, USA, Nov. 2013).

[13] CPUSET: MM: REDUCE LARGE AMOUNTS OF MEMORY BARRIER RELATED DAMAGE V3. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=cc9a6c8776615f9c194ccf0b63a0aa5628235545.

[14] CRAMMING MORE INTO STRUCT PAGE. https://lwn.net/Articles/565097/.

[15] CRANOR, C. D., AND PARULKAR, G. M. The UVM Virtual Memory System. In *USENIX ATC'99* (1999).

[16] DENNING, P. J. Virtual memory. *ACM Computing Survey 2*, 3 (Sept. 1970).

[17] DON'T FORGET TO SET SOFTDIRTY ON FILE MAPPED FAULT. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/?id=9aed8614af5a05cdaa32a0b78b0f1a424754a958&context=40&ignorews=0&dt=0.

[18] FIX TOO LAZY VUNMAP CACHE FLUSHING. https://lkml.org/lkml/2009/6/16/722.

[19] FIX WRONG NUM_POISONED_PAGES IN HANDLING MEMORY ERROR ON THP. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memory-failure.c?id=4db0e950c5b78586bea9e1b027be849631f89a17.

[20] FRAGKOULIS, M., SPINELLIS, D., LOURIDAS, P., AND BILAS, A. Ralational access to Unix kernel data structures. In *EuroSys'14* (Amsterdam, The Netherlands, Apr. 2014).

[21] GANAPATHY, N., AND SCHIMMEL, C. General purpose operating system support for multiple page sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (1998), USENXI ATC '98.

[22] GORMAN, M. Understanding the Linux Virtual Memory Manager. *ISBN 0-13-145348-3* (2004).

[23] GUNAWI, H. S., HAO, M., LEESATAPORNWONGSA, T., PATANA-ANAKE, T., DO, T., ADITYATAMA, J., ELIAZAR, K. J., LAKSONO, A., LUKMAN, J. F., MARTIN, V., AND SATRIA, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *SOCC'14* (Seattle, WA, Nov. 2014).

[24] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *OSDI'14* (Broomfield, CO, Oct. 2014).

[25] HUANG, J., BADAM, A., QURESHI, M. K., AND SCHWAN, K. Unified Address Translation for Memory-Mapped SSDs with FlashMap. In *ISCA'15* (Portland, OR, June 2015).

[26] HUANG, J., ZHANG, X., AND SCHWAN, K. Understanding Issue Correlations: A Case Study of the Hadoop System. In *SOCC'15* (Kohala Coast, HI, Aug. 2015).

[27] IMPROVING THE FUTURE BY EXAMING THE PAST. http://isca2010.inria.fr/media/slides/Turing-Improving_the_future_by_examining_the_past.pdf.

[28] JONES, R. M. Factors affecting the efficiency of a virtual memory. *IEEE Transactions on Computers C-18*, 11 (Nov. 1969).

[29] KADAV, A., AND SWIFT, M. M. Understanding Modern Device Drivers. In *ASPLOS'12* (London, England, UK, Mar. 2012).

[30] KARAKOSTAS, V., GANDHI, J., AYAR, F., CRISTAL, A., HILL, M. D., MCKINLEY, K. S., NEMIROVSKY, M., SWIFT, M. M., AND UNSAL, O. Redundant Memory Mappings for Fast Access to Large Memories. In *ISCA'15* (Portland, OR, June 2015).

[31] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., NORRISH, M., KOLANSKI, R., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal Verification of an OS Kernel. In *SOSP'09* (Big Sky, Montana, Oct. 2009).

[32] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *FAST'13* (San Jose, CA, Feb. 2013).

[33] LI, K., AND CHENG, K. H. A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system. In *Proceedings of the 1990 ACM Annual Conference on Cooperation* (1990), CSC '90.

[34] LINUX 2.6.39 MERGE WINDOW. https://lwn.net/Articles/434637/.

[35] LIST_LRU: PER-NODE LIST INFRASTRUCTURE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=3b1d58a4c96799eb4c92039e1b851b86f853548a.

[36] LOCKLESS (AND PREEMPTLESS) FASTPATHS FOR SLUB. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=8a5ec0ba42c4919e2d8f4c3138cc8b987fdb0b79.

[37] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A Study of Linux File System Evolution. In *FAST'13* (Feb. 2013).

[38] LUMPY RECLAIM. https://lwn.net/Articles/211199.

[39] MEMCG: FIX ZONE CONGESTION. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/vmscan.c?id=d6c438b6cd733834a3cec55af8577a8fc3548016.

[40] MEMCONTROL: DO NOT ACQUIRE PAGE_CGROUP LOCK FOR KMEM PAGES. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memcontrol.c?id=a840cda63e543d41270698525542a82b7a8a18d7.

[41] MEMORY MANAGEMENT IN MEMCACHED. https://code.google.com/p/memcached/wiki/MemoryManagement.

[42] MEMORY-MANAGEMENT TESTING AND DEBUGGING. https://lwn.net/Articles/636549/.

[43] MM ANON RMAP: REPLACE SMAE_ANON_VMA LINKED LIST WITH AN INTERVAL TREE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=bf181b9f9d8dfbba58b23441ad60d0bc33806d64.

[44] MM: AUGMENT VMA RBTREE WITH RB_SUBTREE_GAP. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=d37371870ceb1d2165397dc36114725b6dca946c.

[45] MM: AVOID NULL-POINTER DEREFERENCE IN SYNC_MM_RSS(). https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/memory.c?id=a3a2e76c77fa22b114e421ac11dec0c56c3503fb.

[46] MM, COMPACTION: TERMINATE ASYNC COMPACTION WHEN RESCHEDULING. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=aeef4b83806f49a0c454b7d4578671b71045bee2.

[47] MM: FIX BOUNDARY CHECKING IN FREE_BOOTMEM_CORE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/bootmem.c?id=5a982cbc7b3fe6cf72266f319286f29963c71b9e.

[48] MM: KEEP PAGE CACHE RADIX TREE NODES IN CHECK. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/lib/radix-tree.c?id=449dd6984d0e47643c04c807f609dd56d48d5bcc.

[49] MM: MUNLOCK: MANUAL PTE WALK IN FAST PATH INSTEAD OF FOLLOW_PAGE_MASK(). https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=7a8010cd36273ff5f6fea5201ef9232f30cebbd9.

[50] MM: MUNLOCK: REMOVE REDUNDANT GET_PAGE/PUT_PAGE PAIR ON THE FAST PATH. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=5b40998ae35cf64561868370e6c9f3d3e94b6bf7.

[51] MM: VMSCAN: FIX INAPPROPRIATE ZONE CONGESTION CLEARING. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/vmscan.c?id=ed23ec4f0a510528e0ffe415f9394107418ae854.

[52] MM, X86: SAVING VMCORE WITH NON-LAZY FREEING OF VMAS. ://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=3ee48b6af49cf534ca2f481ecc484b156a41451d.

[53] MMTESTS. https://github.com/gormanm/mmtests.

[54] MORARU, I., ANDERSEN, D. G., KAMINSKY, M., TOLIA, N., RANGANATHAN, P., AND BINKERT, N. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *TRIOS'13* (Farmington, USA, Nov. 2013).

[55] NGUYEN, H. H., AND RINARD, M. Detecting and Eliminating Memory Leaks Using Cyclic Memory Allocation. In *ISMM'07* (Montreal, Quebec, Canada, Oct. 2007).

[56] NUMA: DO NOT TRAP FAULTS ON THE HUGE ZERO PAGE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/huge_memory.c?id=e944fd67b625c02bda4a78ddf85e413c5e401474.

[57] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J., AND ROSENBLUM, M. Fast Crash Recovery in RAMCloud. In *SOSP'11* (Cascais, Portugal, Oct. 2011).

[58] PAGE ALLOCATOR: INLINE BUFFERED_RMQUEUE(). https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=0a15c3e9f649f71464ac39e6378f1fde6f995322.

[59] PAGE ALLOCATOR: USE A PRE-CALCULATED VALUE INSTEAD OF NUM_ONLINE_NODES() IN FAST PATHS. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=62bc62a873116805774ffd37d7f86aa4faa832b1.

[60] PALIX, N., THOMAS, G., SAHA, S., CALVES, C., LAWALL, J., AND MULLER, G. Faults in Linux: Ten Years Later. In *ASPLOS'11* (Newport Bench, California, Mar. 2011).

[61] PERSISTENT MEMORY SUPPORT PROGRESS. https://lwn.net/Articles/640113/.

[62] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *OSDI'14* (Broomfield, CO, Oct. 2014).

[63] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON File Systems. In *SOSP'05* (Brighton, United Kingdom, Oct. 2005).

[64] RADIX-TREE: INTRODUCE BIT-OPTIMIZED ITERATOR. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/lib/radix-tree.c?id=78c1d78488a3c45685d993130c9f17102dc79a54.

[65] RADIX TREES. https://lwn.net/Articles/175432.

[66] RAO, S., HEGER, D., AND PRATT, S. Examining Linux 2.6 Page Cache Performance. In *Ottawa Linux Symposium (OLS'05)* (2005).

[67] READAHEAD: FIX NULL FLIP DEREFERENCE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/readahead.c?id=70655c06bd3f25111312d63985888112aed15ac5.

[68] READAHEAD: FIX SEQUENTIAL READ CACHE MISS DETECTION. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/readahead.c?id=af248a0c67457e5c6d2bcf288f07b4b2ed064f1f.

[69] RED-BLACK TREES. https://lwn.net/Articles/184495.

[70] RUMBLE, S. M., KEJRIWAL, A., AND OUSTERHOUT, J. Log-structured Memory for DRAM-based Storage. In *FAST'14* (Santa Clara, CA, Feb. 2014).

[71] SAXENA, M., AND SWIFT, M. M. Flashvm: Virtual memory management on flash. In *USENIX Annual Technical Conference'10* (2010).

[72] SESHADRI, V., PEKHIMENKO, G., RUWASE, O., MUTLU, O., GIBBONS, P. B., KOZUCH, M. A., MOWRY, T. C., AND CHILIMBI, T. Page Overlays: An Enhanced Virtual Memory Framework to Enable Fine-grained Memory Management. In *ISCA'15* (Portland, OR, June 2015).

[73] SLAB: EMBED MEMCG_CACHE_PARAMS TO KMEM_CACHE. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=f7ce3190c4a35bf887adb7a1aa1ba899b679872d.

[74] SLOCCOUNT. http://www.dwheeler.com/sloccount/.

[75] SLUB: ALTERNATE FAST PATHS USING CMPXCHG_LOCAL. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/slub.c?id=1f84260c8ce3b1ce26d4c1d6dedc2f33a3a29c0c.

[76] SLUB: CHECK FOR PAGE NULL BEFORE DOING THE NODE_MATCH CHECK. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=c25f195e828f847735c7626b5693ddc3b853d245.

[77] SLUB: FAILSLAB SUPPORT. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/failslab.c?id=773ff60e841461cb1f9374a713ffcda029b8c317.

[78] SLUB PAGE ALLOC FALLBACK: ENABLE INTERRUPTS FOR GFP_WAIT. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=caeab084deb61cd2d51cb8facc0e894a5b406aa4.

[79] SOFT DIRTY PTES. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt.

[80] SSD DISCARD COMMAND. https://wiki.archlinux.org/index.php/Solid_State_Drives.

[81] STEINHORST, G. C., AND BATEMAN, B. L. Evaluation of efficiency in a virtual memory environment. In *SICOSIM4* (Oct. 1973).

[82] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. In *SOSP'03* (Bolton Landing, New York, Oct. 2003).

[83] THE LINUX KERNEL ARCHIVES. https://www.kernel.org.

[84] THE ZSWAP COMPRESSED SWAP CACHE. https://lwn.net/Articles/537422/.

[85] THP: ABORT COMPACTION IF MIGRATION PAGE CANNOT BE CHARGED TO MEMCG. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm/compaction.c?id=4bf2bba3750f10aa9e62e6949bc7e8329990f01b.

[86] VIRTUAL MEMORY HISTORY. http://en.wikipedia.org/wiki/Virtual_memory.

[87] VMSCAN: COUNT ONLY DIRTY PAGES AS CONGESTED. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/mm?id=1da58ee2a0279a1b0afd3248396de5659b8cf95b.

[88] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *FAST'15* (Santa Clara, CA, Feb. 2015).

[89] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI'06* (Seattle, WA, Nov. 2006).

[90] YOSHII, K., ISKRA, K., NAIK, H., BECKMAN, P., AND BROEKEMA, P. C. Performance and Scalability Evaluation of 'Big Memory' on Blue Gene Linux. *International Journal High Performance Applications 25*, 2 (May 2011).

[91] ZHANG, H., CHEN, G., OOI, B. C., TAN, K.-L., AND ZHANG, M. In-Memory Big Data Management and Processing: A Survey. *IEEE Transactions on Knowledge and Data Engineering 27*, 7 (Apr. 2015).

Getting Back Up: Understanding How Enterprise Data Backups Fail

George Amvrosiadis
Dept. of Computer Science, University of Toronto
gamvrosi@cs.toronto.edu

Medha Bhadkamkar
Veritas Labs
medha.bhadkamkar@veritas.com

Abstract

In the enterprise world, retaining data backups is the de-facto solution against data loss in the event of catastrophic failures. As backup software evolves to achieve faster backup and recovery times, however, backup systems deploying it become increasingly complex to administer. This complexity stems from optimizations targeted to specific applications, which increase the number of configuration parameters for the system. Still, there is no work in the literature that attempts to study the error characteristics of enterprise backup systems, despite our reliance on the guarantees they provide.

With this study we aim to help researchers and practitioners understand how backup system jobs fail, and identify factors that can be used to predict these failures. Our results are derived from an analysis of data on 775 million jobs, collected from more than 20,000 backup software installations over a span of 3 years. We confirm that trends reported in the software reliability literature also hold for backup systems, such as that the majority of job errors are due to misconfigurations. For the systems in our dataset, we find that error rates remain stable across software versions and over time. To better understand these errors, we investigate the effect of several factors on the system's error rate, such as job sizes and policy complexity, and demonstrate their predictive power for future errors.

1 Introduction

From enterprise organizations to home users, data backups are still the preferred mechanism for the prevention of data loss in the event of catastrophic failures. The guarantees provided by backup software, however, rely on the assumption that periodic backup jobs will complete successfully. Unfortunately, backup software is becoming increasingly complex to configure and manage, as more tuning parameters are introduced to handle diverse application requirements. Recent surveys of CIOs and IT professionals show that 27% of businesses have trouble recovering from backups due to backup jobs not completing successfully [18], and 80% experience challenges managing backup data and configuring the backup software [35]. At the same time, data generation rates increase. In a recent study, we found that full backups are performed as often as every 1-4 days to accommodate data churn [3], leaving less time to repeat failed jobs.

The goal of this study is to help researchers and practitioners understand how backup system jobs fail, and identify the factors that can be used to predict these failures. Our results are based on the analysis of periodic reports collected from customer installations of Veritas NetBackup [37], a commercial backup software product, over the span of 3 years. In total, we studied 775 million jobs from more than 20,000 *backup systems*, i.e. customer installations. Depending on their type, these jobs perform specific operations, such as data backup, recovery, and backup data management (e.g. replication). Jobs are scheduled according to *backup policies*, which are sets of configurable parameters dictating how individual applications should be backed up. For example, VMware policies expose parameters specific to virtual machines, while Oracle policies expose parameters relevant to database instances.

First, we investigate the prevalence of errors in backup systems and their causes. We find that backup system jobs fail frequently, and the majority of errors are attributed to misconfigurations, which confirms a recurrent trend in the literature for other system types [11, 24, 25, 34, 44]. On the bright side, we find that the errors themselves are not diverse, and the 10 most frequent error codes returned by jobs account for more than 78% of job errors. We explain these errors in detail, providing guidelines that designers can leverage to improve the robustness of future backup software.

Next, to understand why errors occur, we study the context in which they appear. Specifically, we study characteristics of the backup system and individual jobs, and the degree to which they affect the frequency and diversity of the occurring errors. For example, we find that factors such as the size of the job, and the configuration complexity of a policy, are likely linked to job failures. On the other hand, factors such as the frequency of configuration updates and software versions, are likely not tied to failures. By quantifying the relationships between

Characteristic	Observation	Section	Previous work
Prevalence	In the average backup system, 15.2% of jobs terminate with an error. The overall job error rate across all systems in our dataset remains stable over time.	4.1	None
Diversity	The 10 most frequent error codes correspond to more than 78% of job errors.	4.2	None
Causes	More than 75% of job errors are due to misconfigurations.	4.3	Similar trends for different applications [11, 24, 25, 27].
Dependence on job properties	Jobs managing (e.g. replicating) existing backup images exhibit the highest error rates.	5.1	None
	Systems with larger jobs tend to experience higher job error rates.	5.2	None
	Policies with more configurable parameters exhibit higher job error diversity.	5.3	None
Dependence on system properties	Backup software and operating system versions have no effect on job error rate.	6.1	None
	The size and load of a backup system is indicative of its job error diversity.	6.2	None
	The rate of configuration changes has no effect on the system's error rate or diversity.	6.3	None
Predictability	The number of daily occurrences of a given error code follows predictable trends, but error inter-arrival times do not.	7.1	None
	Job errors can be fairly accurately predicted based on the occurrence of other error codes. The factors in previous observations, however, make superior predictors.	7.2	None

Table 1: A summary of the most important observations of our study regarding backup system job errors.

each factor and the system's resulting error rate, we aim to provide administrators with rules of thumb that can be consulted during system configuration.

For monitored systems, we receive weekly reports that allow us to observe how these systems evolve over time. We show that the number of daily errors in a backup system follows predictable trends, while error inter-arrival times do not. Finally, we demonstrate that while different error types are correlated, the factors we identify in the observations of our study are better predictors of future error occurrences. We find these results encouraging, suggesting that future backup systems could rely on prediction models to automatically detect and self-heal from errors, without affecting their reliability guarantees.

Table 1 summarizes the most important observations of our study. The remainder of the paper is organized as follows. In Section 2 we present an overview of prior work on backup systems and related work in the software reliability literature. The dataset we use in this study is introduced in Section 3. First, we analyze this data to understand the prevalence and causes of job errors in backup systems in Section 4. Then, we identify job characteristics that can be correlated to higher error rates in Section 5. In Section 6 we examine the relationship between properties of the backup system, and its reported error rate. In Section 7 we evaluate the predictability of errors, and the predictive power of our observations from previous sections for future errors. Finally, we outline the implications of our results in the design of future backup systems in Section 8, and conclude in Section 9.

2 Related work

Little work exists to characterize the operation of backup systems. Park and Lilja [26] used traces of weekly full backup operations from 6 systems to characterize their deduplication ratios. Wallace et al. [39] study the contents and workload of file systems that store data produced by the jobs of backup systems such as NetBackup. Our earlier work [3] identifies and characterizes trends in the configuration and job characteristics of backup systems. Other work [10, 36] has focused on modeling the growth rate of storage capacity in backup systems, but it does not examine the prevalence of storage capacity errors. Overall, none of the aforementioned work looks into the characteristics of errors in the operation of backup systems.

Despite the lack of prior work on backup systems, several studies have examined the characteristics and prevalence of software and administration errors in other system types. In his seminal paper on system faults, Gray [11] reports that 25% of faults in high-end mainframes are due to software errors, while 42% are attributed to administrator errors. Similarly, Patterson et al. [27] observed that 8-34% of failures in telephone networks and Internet systems were due to software, while 51-59% were due to administration errors. Oppenheimer et al. [25] studied the failures of Internet Services and report that failures are due to software errors 33% of the time, while 57% are administration mistakes, and Nagaraja et al. [24] confirm these findings in a user study. Tang et al. [34] report that 16% of high-impact incidents over a three-month period at Facebook were due to misconfigurations, but do not provide a more detailed breakdown. Our study is the first to consider backup systems, but it is worth noting that our breakdown of backup system errors by cause (Figure 5) matches closely the results reported in the literature for other system types.

Due to the prevalence of misconfigurations, several research efforts have put forth ideas to detect, diagnose, and automatically fix these errors. PeerPressure [40] uses a statistical approach and a repository of configurations

Job type	Entries	Function
Backup	604.9 M	Create backup images
Management	105.8 M	Manage (e.g. delete, replicate, migrate) backup images
Snapshot	58.2 M	Create Copy-on-Write data snapshots
Recovery	6.3 M	Recover data from backup images
Total	**775.2 M**	

Table 2: Breakdown of job types in our dataset.

to identify parameter errors in a given configuration. In a similar manner, Yuan et al. [45] identify error root causes using support vector machines to detect anomalous patterns in system call sequences. CODE [46] extends this idea by identifying invariants for configuration access rules. EnCore [47] adopts a learning approach guided by sample configurations, augmented with information on the execution context of the configurations. Chronus [42] retains disk state through periodical checkpoints, and traverses them to detect the configuration changes responsible for the misconfiguration. ConfAid [6] instruments application binaries to trace the configuration entry responsible for the misconfiguration. AutoBash [5, 33] leverages carefully crafted bug-tracking predicates to detect deviations from healthy machines, and uses a speculative OS kernel to find a fix through trial and error and a solution database. Such approaches can benefit greatly from studies outlining the contributing factors to errors. Observations derived from such studies can be used as heuristics for learning approaches, and to prune the state space for data-flow analysis techniques. Furthermore, while these are successful reactive approaches to error occurrence, we also examine the feasibility of proactive approaches, such as error prediction.

3 Dataset description

Our analysis and models are based on data from *telemetry reports* collected from enterprise customer installations of a commercial backup product, Veritas NetBackup [37]. Reports are only collected from customers who opt to participate in the telemetry program, and contain no personal identifiable information. This section outlines the characteristics of our telemetry dataset.

Reported information. Telemetry reports received from customer backup systems contain runtime and configuration information about these systems. This runtime information describes the jobs that run in each system, and whether they completed successfully. The different job types recorded in monitored backup systems are described in Table 2. Jobs run in backup windows, the time and duration of which are configured by the system administrator. When a job fails to complete successfully, it can be retried in the current or next backup window, subject to time availability and priority compared to other jobs. By default, backup jobs are retried once, while other job types are never retried. The job scheduler will also cancel retries of jobs that are scheduled to recur in the current backup window. Our telemetry reports record the error codes returned by failed jobs, but do not allow us to distinguish job retries. In our analysis we use job error codes in conjuction with other system or job characteristics, to investigate why jobs fail.

Dataset size and breadth. Our dataset consists of 1 million telemetry reports, collected over the span of three years, between September 2012 and August 2015. These reports represent 22 minor versions of the backup software, all under the same major version: NetBackup 7.6. More than 20,000 customer installations are represented, across most modern operating systems. Note that this dataset is different from the one used in our earlier work [3]: we have excluded systems that deploy versions of NetBackup earlier than 7.6, since their telemetry reports do not include error information, and extended the dataset by collecting reports for another year.

Architecture. Modern backup systems typically consist of three tiers of operation: a master server, one or more storage servers, and several clients [3]. The *master server* is responsible for scheduling and monitoring backup jobs, while *storage servers* manage the archival of backup images. In smaller systems, the master server can be consolidated with a storage server. In our dataset, 27.9% of backup systems use dedicated storage servers, while the rest consist of one master/storage server.

Monitoring duration. The backup systems described in our study were each monitored for 5.7 months on average, and for a maximum of 34 months. Note that the monitoring time is not always equivalent to the total lifetime of the backup system, as most of these systems were still online at the time of this writing. For some systems, our dataset contains only a single weekly telemetry report. We are unable to distinguish whether these reports correspond to systems with a lifespan of one week or less, refer to failed installations, or are due to reporting errors. Our dataset also contains reports from systems used internally in Veritas for development and quality assurance. As part of pre-processing, we have chosen to exclude almost 5,000 systems from our analysis, which belong to either one of these categories.

4 Error characteristics

In this section, we use our dataset to characterize the frequency of job errors in backup systems. We further investigate the diversity in the error codes returned by in-

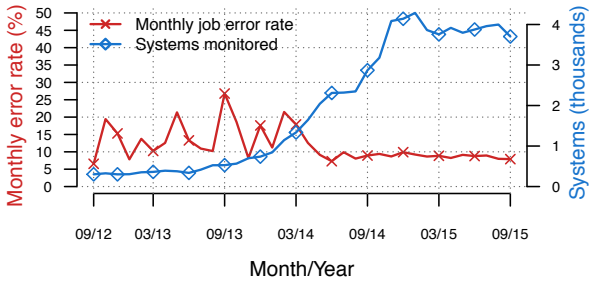


Figure 1: Monthly job error rates in a 3-year time period.

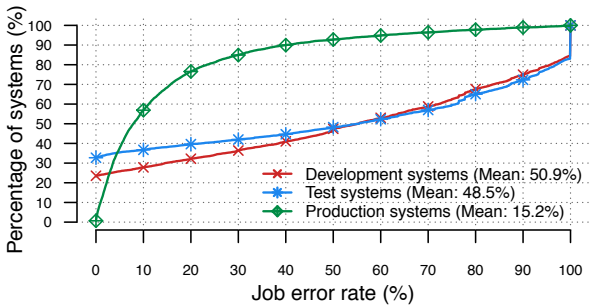


Figure 2: CDFs for the job error rate of individual backup system types.

Category	Version	Jobs	Lifespan	Systems
Production	Stable	176,173	76 days	4,220
Development	Alpha, Beta	5,965	18 days	5,217
Test	Stable	112	10 days	6,066

Table 3: Characteristics of the different types of backup systems in our dataset. The number of jobs and lifespan refer to averages across individual systems.

dividual jobs, and analyze them to find the most popular causes for job errors.

4.1 Prevalence of job errors

After pre-processing, our dataset consists of 775.2 million jobs across 15,503 systems. Of these jobs, 69.4 million (or 8.7%) terminate with an error code indicating a partial, or complete failure. In Figure 1, we show the monthly job error rate across all recorded jobs. Note that as more systems join the telemetry program (right y-axis), the error rate becomes more stable (left y-axis). Since error rates are not improving over time, a better understanding of backup system errors seems imperative.

It is common for customers to test development or stable versions of the backup software. The installations used in this process have radically different error and workload profiles, as will be demonstrated later in the paper. Thus, we have grouped these systems and report results on each system type separately. The different system categories are listed in Table 3. *Production systems* are deployed in the field. They have long lifespans, most still being online at the time of this writing, and are by far the busiest systems. *Development systems* are run by partner organizations that are given early access to new software features through alpha and beta versions. These systems are short-lived, but run workloads with job frequencies comparable to production systems. Finally, *test systems* are customer systems that run stable versions of the software, for the purpose of testing its operation before deploying it in production. These systems are also short-lived, but run a small number of jobs, presumably to test different aspects of the system's operation.

Only 3,243, or 20.9%, of the systems in our dataset exhibit no errors in the time they are monitored. Surprisingly, only 13 of these systems are used in production, while the rest are used for test and development purposes. The three system types also differ with regards to the job error rates they exhibit. Figure 2 shows the empirical cumulative distribution functions (CDFs) for the error rates of systems of each type. The error rate for jobs in production systems is 15.2% on average. Through Kolmogorov-Smirnov tests [23], we confirmed that the job error rates for production systems follow a Weibull distribution, with a shape parameter of $(76.1 \pm 0.9) \times 10^{-2}$ and a scale parameter of 13.0 ± 0.3. Note that the Weibull distribution is typical in systems reliability research, especially for modeling failure rates [30, chapter 2.12]. On the other hand, job error rates for development and test systems approximate the uniform distribution. On average, jobs in development and test systems fail 48.5% and 50.9% of the time, respectively. This is not surprising, as one would expect these systems to be used for testing a variety of scenarios in order to iron out problems before deployment in production. Note that these numbers differ from those in Figures 1 and 9, which aggregate jobs across all backup systems.

Observation 1: *15.2% of jobs terminate with an error in production backup systems. Testing and development systems exhibit up to 3.3 times more errors on average.*

4.2 Error diversity

In NetBackup, there are 1,194 distinct error codes that can be returned when a job fails partially, or completely [38]. Of these error codes, only 333 (or 27.9%) are reported in our dataset.

Backup systems of different types experience different sets of error codes, as jobs fail for different reasons. Specifically, 21 error codes occur only in development systems. These errors are caused by commands failing due to permission or communication issues, invalid in-

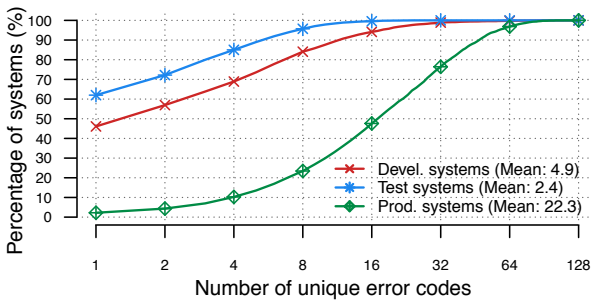


Figure 3: CDFs for the number of unique job error codes for each type of backup system.

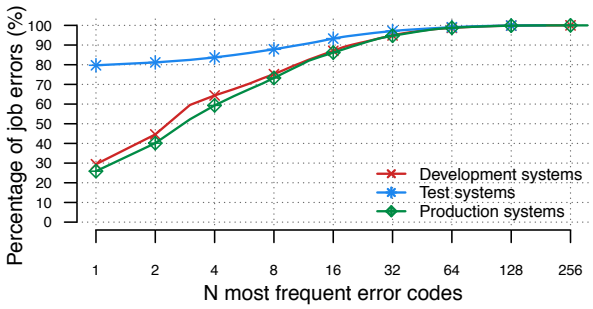


Figure 4: Percentage of job errors due to the N most frequent error codes for different backup system types.

puts, and unsupported or unavailable software features. Similarly, 59 error codes are specific to production systems. These errors occur as a result of communication errors due to offline system components, product licensing issues, misconfigurations, or the inability to fit all scheduled jobs in the specified backup window. Interestingly, the set of error codes that occur in test systems is a strict subset of the errors occurring in both development and production systems. In other words, error codes that occur during testing are likely those that survived development, but they make up only a strict subset of the errors that occur in production.

The set of error codes experienced by individual backup systems can also differ. Figure 3 shows the empirical CDFs for the number of unique error codes exhibited in individual backup systems throughout their lifespan. While average test and development systems experience 2.4 and 4.9 error codes respectively, production systems can exhibit 22.3 error codes on average, almost an order of magnitude difference.

Observation 2: *Production systems experience an order of magnitude more error codes, for three orders of magnitude more jobs run compared to test systems.*

The number of error codes exhibited by a system is not an indicator of the system's job error rate. This is a result of a few error codes being responsible for the vast majority of job errors. In Figure 4, we show the percentage of job errors that correspond to the N most frequent error codes. Jobs are partitioned by system type. Overall, we find that the 10 most frequently occurring error codes correspond to 78.1%, 79.0%, and 89.5% of job errors in production, development, and test systems respectively. Among these errors, half are unique for each system type, while the remaining 5 error codes are common across all systems. Of those, the most common error code denotes a partially successful backup job, which failed to back up some of the requested data because the backup process was unable to access it. Other common error codes occur due to insufficient backup storage capacity, as a result of the backup window being too short to fit all scheduled jobs, or because the storage unit was unreachable.

Observation 3: *The 10 most frequent error codes correspond to 78.1-89.5% of job errors, depending on the system type.*

4.3 Error causes

A smaller number of errors in a backup system implies fewer failure scenarios that need to be accounted for. It is not indicative, however, of the effort necessary to resolve each failure. Since many error codes in NetBackup are overloaded, we are unable to classify them based on their severity. Instead, we categorize error codes based on their cause, and analyze the distribution of job error causes across backup system types.

We manually categorized the error codes that appear in our dataset into three categories, using the troubleshooting guide provided to administrators of the backup software [38]. We classify an error as a *misconfiguration* if it can be attributed to configuration parameters that have been assigned incorrect values, or system components configured in a way that obstructs the correct operation of the backup system. Examples of such errors include: inability to backup data due to incorrect file permissions or locked files, jobs that were aborted because the backup window was configured too short, authentication errors, and jobs that failed due to insufficient backup storage capacity. An error is classified as a *system error* when it describes events that are not directly in the system administrator's control, in both the software and hardware layers. Examples of such errors include: unavailability of system components due to issues at the storage layer, errors at the operating system level, and internal errors of the backup software or applications it interfaces with. Finally, we classify as *informational messages* all error codes that describe non-fatal but unusual scenarios, such as jobs terminated by the administrator, warning messages, and product licensing issues.

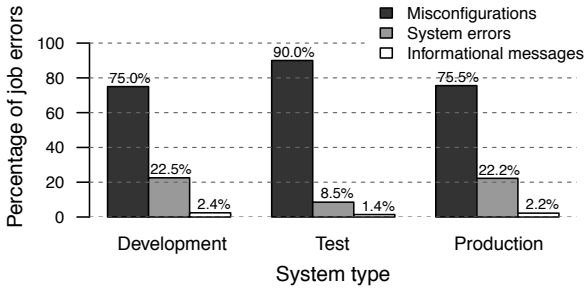


Figure 5: Breakdown of job errors based on their cause.

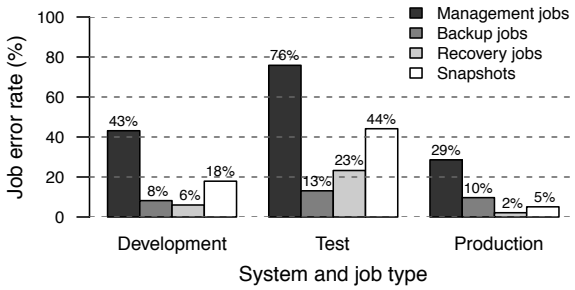


Figure 6: Error rates for different job types, across backup system types.

In Figure 5, we show a breakdown of job errors based on their cause. A separate breakdown is provided for each system type. As is evident, the majority of job errors are due to misconfigurations: 90% in test systems, and over 75% for development and production systems. The remainder of the errors are mostly due to system errors, while 1-2% are informational messages. This result is encouraging because misconfiguration errors usually identify which parameters were incorrectly set, or the constraints that did not apply, causing the error. As a result, they are significantly easier to address compared to system errors, which could start at the hardware level, or any of the storage and operating system layers above. Therefore, we expect that misconfigurations in NetBackup will be a good fit for self-healing approaches. Motivated by this result, we next look into factors that could be used to predict misconfigurations.

Observation 4: *Depending on system type, 75-90% of job errors are attributed to misconfigurations.*

5 Dependence on job properties

To help us prevent job errors, we attempt to identify heuristics that would help us predict them. To derive these heuristics, this section studies the effect of job properties on the manifestation of errors. Specifically, we analyze correlations between job errors, and the characteristics of the affected jobs, such as their type, size, and backup policy type they are part of.

5.1 Job type

We categorize jobs based on the types described in Table 2. Of all the error codes that appear in our dataset, we find that 45.6% are exclusive to a specific job type, while 13.3% are common across all job types. Error codes that are exclusive to specific job types are mostly due to misconfigurations, while error codes that are universal across job types mostly refer to component unavailability, or generic system errors. The remaining error codes are shared between all job types except recovery jobs. Recovery jobs are initiated and controlled directly by users, and only a single error code is logged on error. We do not have sufficient information to correlate data loss due to recovery errors with other job errors.

Recall that in our dataset, the majority of jobs are backups (Table 2). These jobs, however, exhibit low error rates across all system types. In Figure 6, we show the error rates for each job type, across all system types. As expected, production systems have significantly lower job error rates compared to other systems. Overall, management jobs exhibit by far the highest error rates, as high as 29% even in production systems. We find that across system types, management job errors are attributed to operations that attempt to replicate backup images either within the same backup system, or by migrating them to remote backup systems that are administered independently. Most operations that replicate backup images within the same backup system fail due to insufficient storage space, while some fail due to storage device unavailability. On the other hand, most inter-system replication operations fail due to configuration incompatibilities between the systems.

Observation 5: *Management jobs, especially jobs replicating backup images, exhibit the highest error rates.*

5.2 Job size

A limitation of our dataset is that job sizes are not collected individually for each job. Instead, telemetry reports contain the total number of bytes transferred in a given backup system on a given day. We use this information in conjunction with the number of jobs that were executed on the system, to derive the average job size and error rate over the lifespan of the system. Using this data, we study correlations between a system's average job size and its job error rate.

Test and development systems run few jobs that are mostly small in size, and thus there are no significant trends to report. On the other hand, we find that production systems exhibit significant positive correlations be-

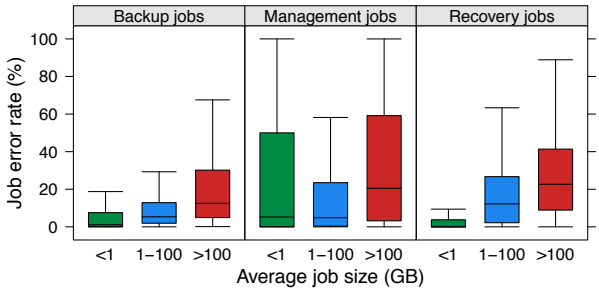


Figure 7: Box plots describing the relationship between the average backup system job size, type and error rate.

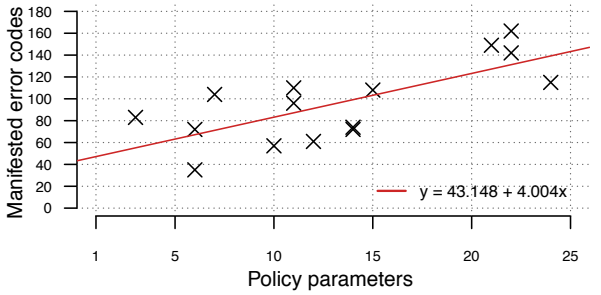


Figure 8: The number of unique error codes due to jobs of a policy type, as a function of the number of policy parameters. Each data point represents a policy type.

tween the average job size in the system, and the system's average job error rate. This holds in different degrees for backup, management, and recovery jobs. In Figure 7, we show box plots describing this relationship for each job type. The lower and upper edges of the boxes represent the 25^{th} and 75^{th} percentiles of job sizes across systems, respectively. The middle bar corresponds to the median, and the whiskers mark the largest and smallest data points in the distribution. Snapshot operations are excluded, as they incur no data transfer. Note that backup and recovery job error rates tend to increase significantly for larger job sizes. Smaller management jobs, however, are characterized by a wider spread. Management jobs of such small sizes are mostly operations used to make changes to the backup system that do not incur data transfers, such as deleting backup images, or examining virtual machine configurations prior to backup. Overall, we find that the most common errors in systems with larger jobs are due to insufficient available storage. In systems with smaller jobs, errors relevant to component unavailability are more common.

Observation 6: *Production systems with larger job sizes tend to experience higher job error rates.*

5.3 Policy type

To achieve consistent backups, applications may require a specific sequence of operations to take place. Thus, backup software offers predefined *policies* tailored to individual applications. For example, a Microsoft Exchange Server policy also backs up the server's transaction log, capturing updates since the backup started. Users can configure policies to specify the characteristics of the backup process, such as the frequency of backup jobs and the retention rate for backed up data. The majority of parameters, however, are policy-specific, e.g. whether to skip unallocated blocks of virtual disks, or back up database redo logs.

We find that the number of unique error codes returned by jobs running under a given policy, is strongly correlated (Pearson's $r_p = 0.73$) with the number of policy parameters exposed to users. In Figure 8 we show the number of unique error codes due to jobs running under a given policy type, as a function of the number of configurable parameters for the policy type. Our results focus only on production systems, which account for 95.9% of jobs. To fit a line to the data we used R's `lm` package [28]. The residual standard deviation (RSD) for the fit is 25.7 error codes, and the line's slope and intercept were chosen with p-values $< 10^{-2}$. The fit indicates that backup system policies are responsible for an additional 4 error codes per configurable parameter, in addition to 43 error codes that are common across policies. Note, however, that we find no correlation between a policy's parameters and its error rate. More complex policies increase the system's error diversity, but not the error count.

Figure 8 reports the number of unique error codes across all systems. Looking at individual backup systems, we also find a strong correlation ($r_p = 0.73$) between the number of policy types deployed, and the number of error codes in a given system. This is important, because we find that individual systems in our dataset deploy 10.7 policy types on average. The best least-squares fit for this relationship is $e = 1.1p + 0.07p^2$, where e is the number of error codes in the system, and p is the number of policy types. The RSD of the fit is 11.7 errors, with coefficient p-values $< 10^{-11}$. While our findings suggest that many error codes do not manifest in a given system, they highlight the relationship between policy configuration and error diversity.

Observation 7: *The number of configurable parameters in a backup policy is positively correlated with the diversity of errors returned from its jobs.*

6 Effect of system setup

We have shown that diversity within the backup system, at the level of policy and job characteristics, can be linked to the number and diversity of job errors. Backup

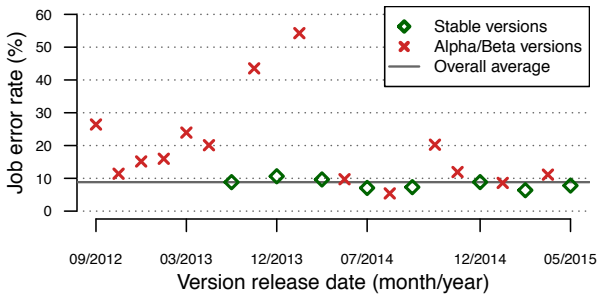


Figure 9: Job error rates across different versions of the backup software.

systems, however, can also differ with regards to the characteristics of the environment in which they are deployed. This section looks into the effect of such factors on job errors: the backup software and operating system versions, the backup system's size and load, and the rate of configuration changes in the system.

6.1 Software components

Overall, we examined backup systems running 22 versions of the backup software. We grouped jobs based on the software version under which they ran, and Figure 9 shows the job error rate for the different versions. We find that the job error rate remains stable across most software versions, approximating the average error rate across all jobs, which is 8.7%. As one would expect, earlier alpha and beta versions of the backup software, running on development systems, demonstrate higher error rates. These error rates are also accentuated by the short lifespan (see Table 3), and the purpose of these systems.

The systems we study are also diverse with regards to the operating environments of their clients, i.e. the machines contributing their data for backup. Backup system clients in our dataset represent several operating system families: Windows, Linux, OS X, AIX, and others. Moreover, they deploy 45 different versions of these operating systems, and many of them often appear in the same backup system. We find no evidence that this client-side heterogeneity can affect the job error rate in the system. This heterogeneity, however, does explain the large number of deployed backup policies (recall Section 5.3). Some policies are tailored to specific operating system families, and we find that backup systems are usually diverse enough to deploy many such policies concurrently, across different clients. Specifically, the majority of backup systems consist of clients deploying at least 6 operating system versions spanning 3 operating system families.

Observation 8: *A system's job error rate remains unaffected across backup software versions and heterogeneity in client operating systems.*

6.2 System size and load

The weekly telemetry reports we collect from customer systems also contain runtime information about the system, such as the number of clients that are active in the system on a given week. We use the number of clients as indication of the system's size, which we find to be non-linearly correlated (Spearman's $r_s = 0.67$) to the system's load. We define the load, or *weekly job rate*, of the system as the average number of jobs that run in the system in a given week. Due to the non-linear relationship between the two quantities, we apply a logarithmic transformation on both quantities before using linear regression to model their relationship. The resulting model is $c = 0.69\,j^{0.6} - 1$, where c is the number of clients, and j is the weekly job rate of the system. The model's RSD is 8.67 clients, with coefficient p-values $< 10^{-8}$. This relationship is likely attributed to clients that join backup policies once they are added to the system, although not all systems follow this tactic [3].

The weekly job rate of a backup system is also correlated ($r_s = 0.56$) to its client heterogeneity. As in Section 6.1, we define a system's client heterogeneity based on the number of different operating system versions represented across its client base. After applying a logarithmic transformation to the weekly job rates, we were able to fit a linear model to the data with RSD of 5.9 operating systems, and coefficient p-values $< 10^{-2}$. Note that while residual errors are too large for prediction purposes, due to high variability in the data, this model indicates the relationship between heterogeneity and system load. The number of client operating systems is expressed as $o = 0.77 \log j + 0.1(\log j)^2$.

Finally, we find that the weekly job rate is indicative ($r_s = 0.62$) of error diversity in the system. Recall that error diversity is defined as the number of unique error codes returned by jobs in a given system. We have fitted a linear model after applying a logarithmic transformation on weekly job rates. The model's RSD is 13.5 errors, with coefficient p-values $< 10^{-15}$. The number of unique error codes in the system is expressed as $e = 0.55(\log j)^2$.

Figure 10 shows the shapes of the curves describing the relationships between a given backup system's weekly job rate, and the three quantities: the number of clients, the number of client operating systems, and the number of unique error codes. We find no evidence, however, that any one of these quantities is indicative of the system's job error rate. In other words, while larger systems should be expected to be busier, more heterogeneous, and exhibit higher error diversity, they shouldn't be expected to have jobs fail at higher rates.

Observation 9: *A backup system's size tends to be indicative of: its load, its clients' heterogeneity, and its error diversity.*

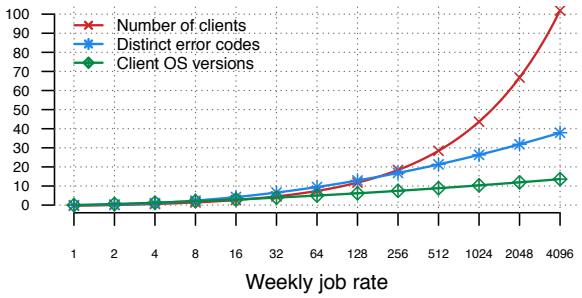


Figure 10: Fits describing the relationships between the average weekly rate of jobs in a backup system and its size, error diversity, and client heterogeneity.

6.3 Configuration changes

For each monitored backup system, we collect weekly statistics on the number of clients, policies, and storage devices configured with the system. To describe the variability for these quantities over time, we estimate their *coefficient of variation*. This metric is defined as the ratio of the standard deviation across our weekly measurements, compared to the overall mean value for the system. As a result, the coefficient corresponds to the fraction of the mean by which a given measurement is expected to deviate, i.e. larger values imply wider spread.

We find that variability in the number of clients of a backup system is strongly positively correlated to the number of backup policies in the system. This implies that the addition and removal of policies in a system coincides with variability in the number of clients. This is likely due to policies that define clients that follow them upon their addition to the system, and due to deleted policies rendering some clients inactive upon their removal.

Lastly, we find no correlation between the variability of a backup system's configuration and its job error rate, or its error diversity. This is attributed to the fact that across all systems, these quantities are characterized by low variability throughout the system's lifespan. Variability in the number of storage devices in the system is also not correlated to any of the previous metrics, for the same reason.

Observation 10: *The rate of configuration changes in a backup system has no effect on its job error rate, or its error diversity.*

7 Error predictability

The majority of errors occurring in a backup system require administrator intervention in order to be resolved (Section 4.3). As a result, we expect these errors to recur over time. In Section 7.1 we examine whether this property describes the errors in our dataset, and whether it can help us predict future errors. Furthermore, jobs in a backup policy execute in a fixed sequence, and thus we expect their success to be dependent on the success of prior jobs. To test this assumption, we examine the existence of dependencies between different error codes in Section 7.2. We further assess the predictive power of factors described in previous sections when forecasting future errors.

7.1 Auto-predictability of error codes

Using the periodic reports we receive for each backup system in our dataset, we can construct time series that reveal information about the timing of error occurrences. Specifically, we have information on the number of individual job error codes that occur in a given backup system, on a specific day. We use this data to construct two types of time series: a series of *error counts*, i.e. the number of times that a given error code occurs daily, and a series of *inter-arrival times*, i.e. the number of days between occurrences of the same error code. Across all systems, we analyze 55,862 error count time series, and 54,461 inter-arrival time series in total [1].

Most of the errors we study are misconfigurations that require administrator intervention in order to be resolved. We expect these errors to be recurrent, and therefore predictable. To test the predictability of our time series, we use the Hurst exponent. This metric is widely used as a measure of the long-term memory of time series [21, 32]. The values of the Hurst exponent, H, range between 0 and 1: $H = 0.5$ indicates a completely uncorrelated series, and $0 < H < 0.5$ corresponds to mean-reverting series, i.e. observations switching between low and high values, gravitating towards the mean. Finally, $0.5 < H < 1$ indicates trend-reinforcing series where future values are likely to be similar to past values. Due to this dependence, the latter category of time series are easier to predict.

Figure 11 shows the CDFs of Hurst values for both types of time series. The values have been estimated using the traditional R/S analysis approach [41]. We find that 90.1% of daily error count series exhibit strong trends ($H > 0.5$). This implies that the occurrences of a given error code tend to follow a trend, instead of varying wildly. The remaining time series score lower because they belong to hosts that have been monitored for shorter periods, and error counts exhibit low variation, if any. Specifically, the majority of series with Hurst exponents less than 0.5 show 1.9 times less variance, and consist of 6.3 times fewer observations. On the other hand, as many as 46.1% of error inter-arrival time series show low Hurst values ($H \leq 0.5$). The reason for this is that these time

[1] The smaller number of inter-arrival series is due to 1,401 error time series that span only two weeks. In those cases, inter-arrival series cannot be defined.

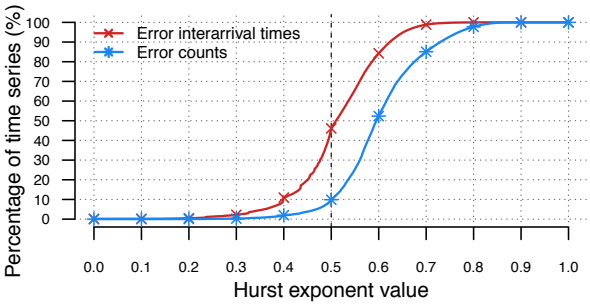


Figure 11: CDFs of Hurst exponent values for time series of error counts and inter-arrival times. Hurst values higher than 0.5 indicate persistent, predictable behavior.

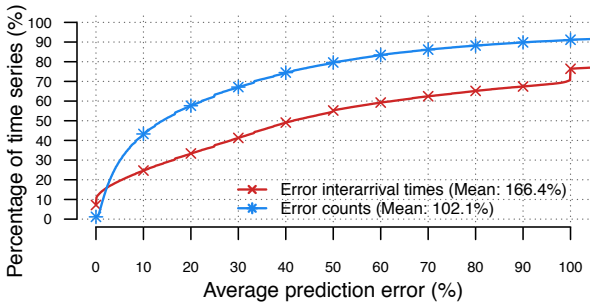


Figure 12: CDFs of the average prediction error of the best auto-regressive and exponential smoothing model fits for each time series.

series are 6.5 times shorter on average, as many errors occur in bursts. As a result, the number of inter-arrival times is smaller than the number of days we have data for, and the resulting trend consists of haphazard spikes.

Observation 11: *The number of daily occurrences of a given error code follows predictable trends, but error inter-arrival times do not.*

While Hurst exponent values reveal trends in time series, they cannot be used to suggest the right method to predict future occurrences. The most widely used forecasting methods for univariate time series are exponential smoothing, and auto-regression. *Exponential smoothing* models make predictions using weighted averages of past observations, with the weights decaying exponentially for older observations [9]. While these models try to capture the trend and seasonality in the data, *auto-regressive* models such as ARIMA [17, chapter 8.9] attempt to explicitly capture correlations with past observations.

For each time series, we estimated the best exponential smoothing and ARIMA model using Akaike's Information Criterion [2], a metric that rewards smaller prediction errors while penalizing models with more parameters. We consider a range of models, such as additive, multiplicative, and damped exponential smoothing. We trained each model using 75% of the available observations, and picked the model that scored the smallest average prediction error on the remaining test data.

In order to compare the prediction accuracy of models fitted to different datasets, we use the mean of the absolute prediction error percentage, a scale-independent error metric [17, chapter 2.5]. Since this metric divides the prediction error for a given observation by its value, it is sensitive to values approaching zero, and undefined for zero values. Therefore, we scaled all time series by adding the same constant prior to model fitting. In Figure 12, we show the CDFs of the average prediction error for each time series of error inter-arrival times, and error counts. A point (x,y) in the graph indicates that y% of time series predict future values with at most x% error.

We find that for the majority of time series, error inter-arrivals cannot be predicted with an error smaller than 42% on average. Error counts are more predictable, as their Hurst exponent values imply, with the majority of time series exhibiting less than 14% error in predictions. Both distributions are long-tailed, however, and predictions can exhibit errors as high as 2400% (or 166% on average) for inter-arrival series, and 886% (or 102% on average) for error counts. Since errors are expressed relative to true values, large errors are expected for short time series with high variability. In any case, these results indicate that job errors cannot be reliably predicted across many systems using only historical information on error occurrence. In the following subsection, we investigate the predictive power of other factors that affect the job error rate, as a means of increasing the prediction accuracy of our models.

Observation 12: *Forecasting methods that take into account only historical information of error occurrences fail to reliably predict future occurrences.*

7.2 Inter-error dependencies

Backup system policies define a fixed sequence of jobs that need to be scheduled before the assigned data has been backed up in a consistent manner. As a result, we expect the success of backup system jobs to be tied to the success of earlier jobs. To investigate whether dependencies exist between individual error codes in such a manner, we need to measure correlations between the time series of every possible error code pair in a system. Unfortunately, doing so manually is infeasible, because metrics that summarize cross-correlation as a number make assumptions about the data that must be verified through rigorous inspection of the time series involved. On the other hand, machine learning models can be used to discover patterns without being as restrictive.

We choose to model our data using random forests of decision trees for individual error codes, where the output for each forest determines whether an error code will occur on a given day, given inputs on the occurrence of other error codes on the same day, and in the last week. Random forests are groups of decision trees, each of which is built using a random subset of the input features and the training data. The output is decided by taking a simple majority vote over all the trees. Due to their construction, random forests are more robust against bias and over-fitting, compared to individual decision trees.

We trained individual random forests for every error code using daily data across all customer production systems, in order to detect trends that are independent of the administrative decisions made in individual systems. We use R's `randomForest` package [7], with 500 trees per forest, and we split our data so that 25% of the observations are reserved for testing the generated model. We measure model accuracy by computing the model's ROC curve [31], which is a function of the model's classification performance given its sensitivity, and then calculating the area under the curve (AUC) [12]. This number represents the model's ability to distinguish true positives without incurring additional false positives, as the sensitivity of the classification is varied. An accuracy of 50% represents the random classifier, while 100% indicates the perfect model that avoids false classifications entirely.

Overall, we find that random forest models trained using historical error data perform fairly well, achieving AUC values in the range of 70-83%, with a median of 76%. To find the features that contribute the most to the model's accuracy, the values of each feature are permuted, i.e. fuzzed, and the overall drop in the model's accuracy is measured. This approach helps us identify relationships between error codes. As a case in point, errors indicating that backups were rejected because the scheduling window closed are usually preceded by informational messages indicating that the administrator has dequeued or killed active jobs. Another example is that of errors indicating failures due to insufficient storage space, which are usually preceded by errors indicating that some, or no files have been backed up successfully.

While our results indicate clear dependencies, it is hard to imagine that the accuracy of the presented models will be sufficient to make predictions in production systems. To further test the validity of our observations, we extended our training data to contain features inspired from the observations made throughout this paper; the full list of features is shown in Table 4. Retraining our random forests with these extra features leads to consistently increased accuracy across all models. Specifically, the AUC values of these models lie in the range of 77-90%, with a median of 83%. We find that the most important feature in 44% of the retrained models is one of the features introduced in our earlier observations. In addition to that, 98% of the models contain at least two such features in their five most important features. It is worth noting that the features of highest significance across all models are the number of jobs, and the average number of parameters (i.e. complexity) of active policies.

Feature Description	Type
Occurrence of error code *i* today	Boolean
Past week occurrences of error code *i*	Numeric
Average number of jobs daily	Numeric
Occurrence of backup jobs today	Boolean
Occurrence of management jobs today	Boolean
Occurrence of recovery jobs today	Boolean
Average number of active policy parameters	Numeric
Average system job size	Numeric

Table 4: Features used in the training of the random forests. The first two feature types refer to all error codes except the one classified. The other features are specific to the system's operation and configuration.

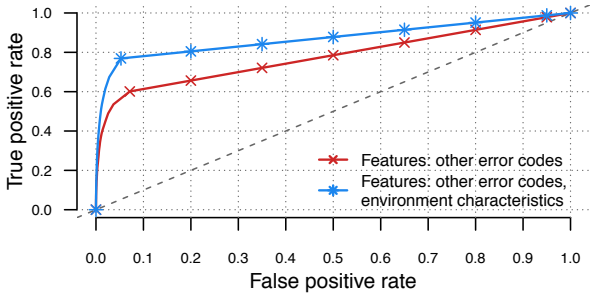


Figure 13: ROC curves for random forests predicting the occurrence of the error code indicating that a storage server is unreachable.

Figure 13 shows the ROC curves of two random forests built to predict the error code indicating that a storage server is unreachable to the backup system. The AUC for the model trained only using historical error data is 77.8%, and Figure 13 shows that it incurs a high rate of false positives once the true positive rate is adjusted above 60%. On the other hand, for the model taking into account factors from our study's observations, the AUC increases to 85.1% and a true positive rate of 80% is possible while retaining a low false positive rate.

Observation 13: *Most errors can be fairly accurately predicted based on the occurrence of other error codes. The factors in previous observations shown in Table 4, however, make superior predictors.*

8 Toward more resilient backup systems

This section outlines the implications of our results on the design of future backup systems. We identify four major topics with potential to increase the resilience of backup systems to job failures.

Error prediction. Very little work exists in the literature on the application of predictive models on backup systems. Chamness [10] and Vaughn et al. [36] use regression to forecast shortages in storage capacity. Ma et al. [22] use Naive Bayes to estimate the failure probability of hard disks, and RAID arrays. We have applied predictive models on job errors, and our results suggest that their occurrences follow predictable patterns. Our models, however, are constructed by simply applying well-known learning methods. We believe that more sophisticated approaches have the potential to yield higher levels of accuracy that would allow administrators to be proactive about errors. While we have trained our models to detect patterns that are universal across all systems in our dataset, prediction models used in individual systems could benefit from biases toward system characteristics.

Configuration automation. While self-healing systems automatically discover and correct faults, they do not improve the odds that they will not occur again in the future. A small body of prior work attempts to remedy this through system autoconfiguration. AutoBash [33] leverages a speculative OS kernel to fix misconfigurations. Chronus [42] makes use of checkpointing and rollback to detect the last working configuration. KarDo [20] takes a machine learning approach to learn the steps required to resolve the issue from a crowd-sourced collection of solutions, supporting heterogeneous environments as well. NetPrints [1] collects examples of good and bad network configurations, and generates a decision tree to determine the set of configuration changes required to transition the system to a good state. One of the challenges that these techniques face, is their timeliness in providing solutions. In some contexts, such as that of backup systems, allowing misconfigurations to remain latent can adversely affect the system's availability. Using the results from studies such as this one, these tools could improve their efficiency by narrowing down the list of potential errors. This filtering could be achieved based on the probability of occurrence for individual error codes, tracking the root cause faster.

Configuration validation. Research on misconfiguration errors mainly focuses on detecting, diagnosing, and troubleshooting these issues. While this provides a remedy after the fact, it does not spare users from the frustration of dealing with these errors in the first place. To address this issue, a line of research focuses on improving the design of configuration interfaces, and making systems more resilient in the face of misconfigurations. Some of the existing work includes the extraction of configuration parameter types [29], statically analyzing code to infer configuration variable constraints [43], permuting valid configuration settings [19], testing operator actions in sandboxes [24], storing application state to seamlessly undo and replay events in the case of errors [8], and using a declarative language to express configuration specifications [15]. Our findings in this study pinpoint configuration variables typically linked to error frequency or diversity in backup systems, and identify other factors as having no effect. Furthermore, our previous work [3] suggests that configuration parameters of these systems are often set to default values. We expect that using these observations as heuristics will assist future work in improving the efficiency of configuration correctness proofs.

Work reduction. One of the most common errors across systems indicates jobs being aborted because the scheduled window for backups was too short. While this error could be resolved by increasing the length of the backup window, it may not be feasible to do so. Today, capacities of hard drives are increasing faster than their data transfer speeds, causing near-line hard drives to take 1.7 hours to access a single terabyte of data [13]. To avoid these delays, it is worth looking into approaches that would allow the amount of maintenance work to be reduced. This could be achieved via content-aware backups that exclude temporary or unimportant files [14, 16], or by piggybacking on other ongoing I/O [4].

9 Conclusion

We have analyzed an extensive dataset from customer backup systems, consisting of 775 million jobs. We find that job errors are prevalent in backup systems, mostly due to misconfigurations. Fortunately, the errors that occur most frequently are not diverse in nature, with 10 error codes accounting for over 78% of job errors.

We identified factors that are responsible for job failures in backup systems, such as the job's type, size, and the complexity of the backup policy that issued the job. We also highlight factors with little effect on job failures, such as software characteristics, system size and load, and configuration variability. We show that the most influential factors make superior predictors for future failures compared to historical data on job errors.

We hope that our observations can be used as guidelines in the design of more robust backup software. We identify four promising directions for future work: error prediction, configuration automation and validation, and work reduction. We believe these features will be necessary in the face of modern data growth rates, which increase the time required to finish backups, leaving less time to rerun failed jobs.

Acknowledgments

The study would not be possible without the telemetry data collected by Veritas' NetBackup team. We especially thank Liam McNerney for his tireless assistance in understanding and extending the telemetry collection infrastructure. We also thank the four anonymous reviewers and our (recurring) shepherd, Fred Douglis, for their invaluable help in improving our paper. Finally, we would like to thank Bruce Montague, CW Hobbs, Ashwin Kayyoor, Vish Janakiraman, Henry Aloysius, Steve Vranyes, and all other members of Veritas Labs for their feedback during earlier stages of our study.

References

[1] AGGARWAL, B., BHAGWAN, R., DAS, T., ESWARAN, S., PADMANABHAN, V. N., AND VOELKER, G. M. NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation* (2009), NSDI'09, pp. 349–364.

[2] AKAIKE, H. Information theory and an extension of the maximum likelihood principle. In *Proc. of the International Symposium on Information Theory* (1973).

[3] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying Trends in Enterprise Data Protection Systems. In *Proc. of USENIX Annual Technical Conference* (2015), ATC'15, pp. 151–164.

[4] AMVROSIADIS, G., BROWN, A. D., AND GOEL, A. Opportunistic Storage Maintenance. In *Proc. of the 25th Symposium on Operating Systems Principles* (2015), SOSP'15, pp. 457–473.

[5] ATTARIYAN, M., AND FLINN, J. Using Causality to Diagnose Configuration Bugs. In *Proc. of the USENIX Annual Technical Conference* (2008), ATC'08, pp. 281–286.

[6] ATTARIYAN, M., AND FLINN, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation* (2010).

[7] BREIMAN, L., CUTLER, A., LIAW, A., AND WIENER, M. `randomForest`: Breiman and Cutler Random Forests for Classification and Regression. https://cran.r-project.org/package=randomForest, Oct. 2015.

[8] BROWN, A. B., AND PATTERSON, D. A. Undo for Operators: Building an Undoable e-Mail Store. In *Proc. of the USENIX Annual Technical Conference* (2003), ATC'03.

[9] BROWN, R. G. *Exponential Smoothing for Predicting Demand*. Arthur D. Little Inc., 1956.

[10] CHAMNESS, M. Capacity Forecasting in a Backup Storage Environment. In *Proc. of the International Conference on Large Installation System Administration* (2011).

[11] GRAY, J. Why Do Computers Stop And What Can Be Done About It?, 1985.

[12] HANLEY, J. A., AND MCNEIL, B. J. The Meaning and Use of the Area under a Receiver Operating Characteristic (ROC) Curve. *Radiology 143*, 1 (Apr. 1982), 29–36.

[13] HETZLER, S., AND COUGHLIN, T. Touch Rate: A metric for analyzing storage system performance, 2015.

[14] HILDRUM, K., DOUGLIS, F., WOLF, J. L., YU, P. S., FLEISCHER, L., AND KATTA, A. Storage Optimization for Large-scale Distributed Stream-processing Systems. *Trans. Storage 3*, 4 (Feb. 2008), 5:1–5:28.

[15] HUANG, P., BOLOSKY, W. J., SINGH, A., AND ZHOU, Y. ConfValley: A Systematic Configuration Validation Framework for Cloud Services. In *Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems* (2015), EuroSys '15.

[16] HUGHES, D., AND FARROW, R. Backup Strategies for Molecular Dynamics: An Interview with Doug Hughes. *Proc. USENIX ;login: 36*, 2 (Apr. 2011), 25–28.

[17] HYNDMAN, R. J., AND ATHANASOPOULOS, G. *Forecasting: Principles and Practice*. oTexts, 2015.

[18] IRON MOUNTAIN. Data Backup and Recovery Benchmark Report. http://www.ironmountain.com/Knowledge-Center/Reference-Library/View-by-Document-Type/White-Papers-Briefs/I/Iron-Mountain-Data-Backup-and-Recovery-Benchmark-Report.aspx, 2013.

[19] KELLER, L., UPADHYAYA, P., AND CANDEA, G. ConfErr: A tool for assessing resilience to human configuration errors. In *Proc. of the IEEE International Conference on Dependable Systems and Networks* (2008).

[20] KUSHMAN, N., AND KATABI, D. Enabling Configuration-independent Automation by Non-expert Users. In *Proc. of the USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10.

[21] LELAND, W. E., TAQQU, M. S., WILLINGER, W., AND WILSON, D. V. On the Self-similar Nature of Ethernet Traffic. *IEEE/ACM Trans. Netw. 2*, 1 (Feb. 1994), 1–15.

[22] MA, A., DOUGLIS, F., LU, G., SAWYER, D., CHANDRA, S., AND HSU, W. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proc. of the USENIX Conference on File and Storage Technologies* (2015), FAST'15, pp. 241–256.

[23] MASSEY JR., F. J. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association 46*, 253 (1951), 68–78.

[24] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation* (2004), OSDI'04.

[25] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proc. of the USENIX Symposium on Internet Technologies and Systems* (2003), USITS'03.

[26] PARK, N., AND LILJA, D. J. Characterizing Datasets for Data Deduplication in Backup Applications. In *Proc. of the IEEE International Symposium on Workload Characterization* (2010), IISWC'10.

[27] PATTERSON, D., BROWN, A., BROADWELL, P., CANDEA, G., CHEN, M., CUTLER, J., ENRIQUEZ, P., FOX, A., KICIMAN, E., MERZBACHER, M., OPPENHEIMER, D., SASTRY, N., TETZLAFF, W., TRAUPMAN, J., AND TREUHAFT, N. Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Tech. Rep. UCB/CSD-02-1175, EECS Department, University of California, Berkeley, Mar. 2002.

[28] R DOCUMENTATION. Fitting linear models. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/lm.html.

[29] RABKIN, A., AND KATZ, R. Static Extraction of Program Configuration Options. In *Proc. of the International Conference on Software Engineering* (2011), ICSE'11.

[30] RAUSAND, M., AND HØYLAND, A. *System Reliability Theory: Models, Statistical Methods and Applications (Second Edition)*. Wiley-Interscience, 2003.

[31] Receiver operating characteristic. https://en.wikipedia.org/wiki/Receiver_operating_characteristic.

[32] RISKA, A., AND RIEDEL, E. Disk drive level workload characterization. In *Proc. of the USENIX Annual Technical Conference* (2006), ATC'06.

[33] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proc. of the 21st ACM Symposium on Operating Systems Principles* (2007), SOSP'07, pp. 237–250.

[34] TANG, C., KOOBURAT, T., VENKATACHALAM, P., CHANDER, A., WEN, Z., NARAYANAN, A., DOWELL, P., AND KARL, R. Holistic Configuration Management at Facebook. In *Proc. of the 25th Symposium on Operating Systems Principles* (2015), SOSP'15, pp. 328–343.

[35] VANSON BOURNE. Virtualization Data Protection Report 2013 – SMB edition. http://www.dabcc.com/documentlibrary/file/virtualization-data-protection-report-smb-2013.pdf, 2013.

[36] VAUGHN, C., MILLER, C., EKENTA, O., SUN, H., BHADKAMKAR, M., EFSTATHOPOULOS, P., AND KARDES, E. Soothsayer: Predicting Capacity Usage in Backup Storage Systems. In *Proc. of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Oct. 2015), MASCOTS'15, pp. 208–217.

[37] VERITAS TECHNOLOGIES. Veritas NetBackup 7.7. https://www.veritas.com/product/backup-and-recovery/netbackup, Sept. 2015.

[38] VERITAS TECHNOLOGIES. Veritas NetBackup Status Codes Reference Guide (Release 7.6). https://www.veritas.com/support/en_US/article.DOC6471, Sept 2015.

[39] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of Backup Workloads in Production Systems. In *Proc. of the USENIX Conference on File and Storage Technologies* (2012), FAST'12.

[40] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation* (2004), OSDI'04.

[41] WERON, R. Estimating long-range dependence: finite sample properties and confidence intervals. *Physica A Statistical Mechanics and its Applications 312* (Sept. 2002), 285–299.

[42] WHITAKER, A., COX, R. S., AND GRIBBLE, S. D. Configuration Debugging As Search: Finding the Needle in the Haystack. In *Proc. of the USENIX Symposium on Opearting Systems Design and Implementation* (2004), OSDI'04.

[43] XU, T., ZHANG, J., HUANG, P., ZHENG, J., SHENG, T., YUAN, D., ZHOU, Y., AND PASUPATHY, S. Do Not Blame Users for Misconfigurations. In *Proc. of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP'13, pp. 244–259.

[44] YIN, Z., MA, X., ZHENG, J., ZHOU, Y., BAIRAVASUNDARAM, L. N., AND PASUPATHY, S. An empirical study on configuration errors in commercial and open source systems. In *Proc. of the 23th Symposium on Operating Systems Principles* (2011), SOSP'11, pp. 159–172.

[45] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated Known Problem Diagnosis with Event Traces. In *Proc. of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), EuroSys'06, pp. 375–388.

[46] YUAN, D., XIE, Y., PANIGRAHY, R., YANG, J., VERBOWSKI, C., AND KUMAR, A. Context-based Online Configuration-error Detection. In *Proc. of the USENIX Annual Technical Conference* (2011), ATC'11.

[47] ZHANG, J., RENGANARAYANA, L., ZHANG, X., GE, N., BALA, V., XU, T., AND ZHOU, Y. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14.

SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision

Mohammadreza Najafi[†], *Mohammad Sadoghi*[‡], *Hans-Arno Jacobsen*[§]
[†]*Technical University Munich*
[‡]*IBM T.J. Watson Research Center*
[§]*Middleware Systems Research Group*

Abstract

There is a rising interest in accelerating stream processing through modern parallel hardware, yet it remains a challenge as how to exploit the available resources to achieve higher throughput without sacrificing latency due to the increased length of processing pipeline and communication path and the need for central coordination. To achieve these objectives, we introduce a novel top-down data flow model for stream join processing (arguably, one of the most resource-intensive operators in stream processing), called SplitJoin, that operates by splitting the join operation into independent storing and processing steps that gracefully scale with respect to the number of cores. Furthermore, SplitJoin eliminates the need for global coordination while preserving the order of input streams by re-thinking how streams are channeled into distributed join computation cores and maintaining the order of output streams by proposing a novel distributed punctuation technique. Throughout our experimental analysis, SplitJoin offered up to 60% improvement in throughput while reducing latency by up to 3.3X compared to state-of-the-art solutions.

1 Introduction

Scalable stream processing is an integral part of a growing number of data management applications such as real-time data analytics [1], algorithmic trading [2], intrusion detection [3], and targeted advertising [4]. These latency-sensitive and throughput-intensive applications have motivated database research to seek new avenues for accelerating data management operations in general and stream processing in particular. These new approaches have adopted heterogeneous architectures (*e.g.*, GPUs and Cell processors) [5, 6, 7, 8], multi-core architectures [9, 10, 11, 12, 13], and Field Programmable Gate Arrays (FPGAs) [2, 14, 15, 16, 17, 18, 19, 20] for stream processing acceleration.

Besides leveraging hardware acceleration, coping with the high-velocity of unbounded incoming streams has forced the stream operation model to shift away from the traditional "store and process" model that has been prevalent in database systems for decades. However, the mindset of sequential stream join processing (or constructing lengthy processing pipelines) and, essentially, thinking of a stream as a sliding window (or a long chain of sequentially incoming tuples to resemble database relations) has continued to shape the way stream processing is carried out today, even on low-latency and high-throughput stream processing platforms.

Stream Join Challenges: To mitigate the challenges imposed by unbounded streams, with respect to both processing and space constraints, data streams are conceptually seen as bounded sliding windows of tuples (*i.e.*, simulating a relation). Sliding windows are defined as a function of time or as a fixed number of tuples. Once the sliding window abstraction is set (*i.e.*, tuples are admitted for processing), the stream join semantics over the windows are identical to the traditional join semantics in relational database systems.

Although the sliding window provides a robust abstraction to deal with the unboundedness of data streams [6, 9, 10, 11, 12, 13, 16, 19], it remains a challenge to improve parallelism within stream join processing, especially, when leveraging many-core systems. For example, a single sliding window could conceptually be divided into many smaller sub-windows, where each sub-window could be assigned to a different join core.[1] However, distributing a single logical stream into many independent cores introduces a new coordination challenge: to guarantee that each incoming tuple in one stream is compared exactly once with all tuples in the other stream.

The coordination challenge is addressed by handshake join [9] that transforms the stream join into a bi-directional data flow problem: tuples flow from left-to-right (for S stream) and from right-to-left (for R stream)[2] and pass through each join core. The bi-directional data flow ensures that every tuple is compared exactly once by design (shown in Figure 1). This new data flow model offers greater

[1] A join core is an abstraction that could apply to a processor's core, a compute node in a cluster or a custom-hardware core on FPGAs (*e.g.*, [9, 19, 16, 10].)

[2] The join operator is performed on two streams which we refer to as S and R streams.

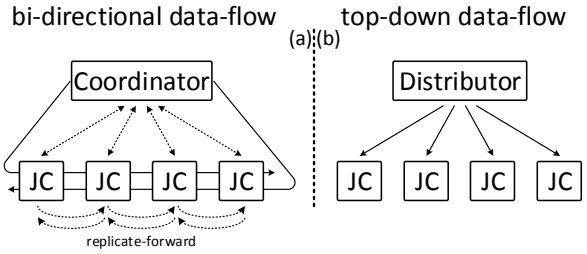


Figure 1: Stream join data flow models (*JC* stands for join core): (a) bi-directional and (b) top-down.

processing throughput by increasing parallelism, yet suffers from latency increase since the processing of a single incoming tuple requires a sequential flow through the entire processing pipeline. To improve latency, yet another central coordination is introduced to fast-forward tuples through the linear chain of join cores in the low-latency handshake join [10] (the coordination module is depicted in Figure 1). For each stream, in order to reduce latency, once a tuple reaches a join core, the tuple is replicated and forwarded to the next core before the join computation is carried out [10].

Generally speaking, any central coordination prohibitively limits the scalability of processing as the degree of parallelism is increased (*cf.* Amdahl's Law); and, central coordination is required in [6, 9, 10, 19]. For example, the coordinator must explicitly send expiration messages to each join core as new tuples enter and old tuples leave each sub-window assigned to a join core. Furthermore, the flow of new and expired tuples in and out of cores is further complicated if tuples are additionally replicated and fast-forwarded [6, 10]. Consequently, the neighboring cores must explicitly communicate (in addition to communicating with the global coordinator) and, in fact, all tuples from both streams actually pass through this communication channel [9, 10]. Moreover, an explicit knowledge of the underlying hardware is required (that may not even be available in virtual machine settings), and one must rely on a complex optimal assignment of the join cores to physical cores to reduce the NUMA-effect by reducing the size of communication paths between neighboring cores [9, 10].

Problem Statement: In this paper, we tackle two main shortcomings of existing stream join processing architectures: the sequential operation model (*i.e.*, "store" and "process") and the linear data flow model (*i.e.*, "left-to-right" and "right-to-left" flows). We propose SplitJoin, the first step in re-thinking the stream join operation model, which is built on the implicit assumption that storage of newly incoming data, whether stored in a relation or a memory buffer, must always precede processing. Instead, we abstract the computation steps as two independent and concurrent steps, namely, (i) "storage" and (ii) "processing".[3] This new splitting abstraction of join cores enables unprecedented scalability by allowing the system to distribute the execution across many independent storage cores[4] and processing cores. Second, we change the way tuples enter and leave the sliding windows, namely, by dropping the need to have separate left and right data flows (*i.e.*, bi-directional flow). SplitJoin introduces a novel top-down data flow (*i.e.*, a single flow), where incoming tuples (from both streams) are simply arriving via the same path downstream (preserving input stream order), while the join results are further pushed and merged downstream using a novel relaxed adjustable punctuation (RAP) technique (preserving the output stream order). Unlike recent advances in stream join processing [6, 9, 19, 10], SplitJoin does not rely on central coordination for propagating and ordering the input/output streams.

SplitJoin's top-down data flow trivially satisfies the ordering of incoming tuples and eliminates the in-flight race condition between the left and right streams as tuples travel from one core to the next. Unlike existing approaches [9, 10], the top-down flow also eliminates the need for communication between the join cores. In SplitJoin, the top-down flow is realized using a distribution tree for routing incoming tuples into their corresponding sub-window that addresses the scaling issues of adding new join cores. The adopted distribution mechanism nicely fits into the coordination-free protocol of SplitJoin for distributing new tuples to both storage and processing cores. For example, all join cores receive the newly incoming tuples (achieving the desired expedited delivery, without the linear forwarding used in [10]), while only one storage core stores the new tuple. Both the storage and eviction of tuples to and from cores are done in a round-robin fashion; thus, naturally, in the same order that cores store a new tuple, they evict their oldest tuple (again, without any explicit coordination). This can be generalized to batches of tuples instead of a single tuple as well.

SplitJoin has provably lower runtime complexity as compared to state-of-the-art parallel distributed join algorithms [9, 10]. SplitJoin exhibits an overall system latency of $O(\log_b k)$, where k is the number of join cores and b is the branching factor of the distribution tree. In contrast, the state-of-the-art handshake join has $O(k)$, while the original version resulted in an $O(n)$ latency, where n is the number of tuples a window can hold ($k \ll n$) [9, 10].

SplitJoin's coordination-free distribution also lends itself to a simpler resiliency against failures; for example, core failures do not halt or disrupt the entire join computation and affect only the failed nodes (the loss is limited to only failed nodes). In contrast, in a linear left-to-right data flow, if any cores fails, then, on average, half of the cores may not receive any data.

SplitJoin is comprised of the following core components: a distribution network, a set of independent join cores, and a

[3] In relational databases, tuples are first stored in relations prior to being processed (*e.g.*, performing a join) while in a stream join, the incoming tuples are first processed and subsequently stored in sliding windows [21].

[4] A storage core is an abstraction for an in-memory sliding window, tightly coupled with a join core.

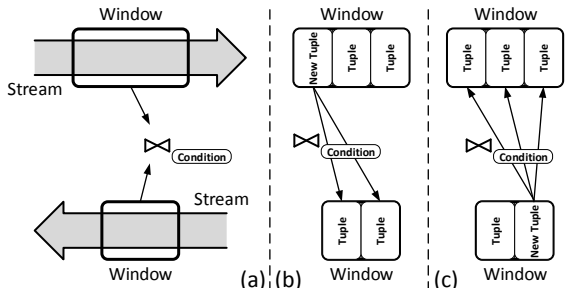


Figure 2: Sliding window concept in stream join.

result gathering network. The distribution network broadcasts incoming tuples to the set of join cores in a scalable way. The actual join computation is carried out by each join core independently, and subsequently, the joined tuples are pushed down to the result gathering network. The result gathering network is further responsible to ensure the correct ordering of the joined tuples by using the punctuation marks produced by the join cores.

In this paper, we make the following contributions:

(1) we propose SplitJoin, a novel scalable stream join architecture that is highly parallelizable and removes inter-core communications and dependencies,

(2) we introduce a new splitting abstraction in SplitJoin to "process" and "store" incoming data streams concurrently and independently,

(3) we propose a top-down data flow model to achieve a coordination-free protocol for distributing and parallelizing stream join processing,

(4) we develop a distribution tree with logarithmic access-latency for routing of incoming data to storage and processing cores, while preserving the ordering of incoming tuples,

(5) we design a coordination-free protocol that does not rely on global knowledge to produce ordered join output streams by proposing a relaxed adjustable punctuation (RAP) technique with tunable precision, and

(6) we conduct an extensive analytical and experimental study of SplitJoin as compared to existing state-of-the-art solutions.

2 Preliminaries

The relational join (theta join) between two non-stream relations R and S, defined as $R \bowtie_\theta S$, produces the set of all resulting pairs (r,s), which satisfy the join condition $\theta(r,s)$ and $r \in R$, $s \in S$. Extending this definition to stream join implies the same join processing semantics with the exception that streams, unlike relations, are unbounded. To mitigate the challenge of unbounded streams, with respect to both processing and storage limitations, streams are conceptually seen as bounded sliding windows of tuples, as shown in Figure 2. The size of these windows are defined as a function of time or number of tuples, referred to as time-based or count-based windows, respectively.

Figure 3 shows the traditional architecture of a join

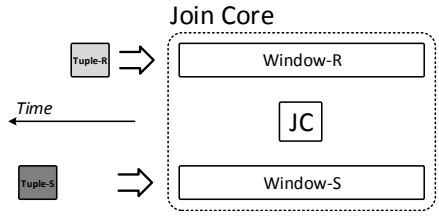


Figure 3: Traditional stream join architecture.

operator that receives Tuple-R and Tuple-S from streams R and S, respectively. *JC* stands for join core, which performs the join operation. To process the tuples shown in the figure, Tuple-R is inserted into *Window-R*, then it is evaluated against all existing tuples in *Window-S* and the join results are returned. Similarly, Tuple-S is inserted into *Window-S* and the same join procedure is applied.

3 Related Work

Work related to our approach can be broadly classified into stream join algorithms [6, 9, 10, 16, 21, 22], or more generally speaking, stream processing in software [23, 24, 25], approaches to performance-optimize stream processing through emerging hardware mechanisms [26], in particular, through FPGA-based acceleration [15, 17], but also, through GPUs and processor-based I/O processing innovations [8]. The survey [27] covers other related work and topics including concepts such as ordering in stream join. SplitJoin can be incorporated in any of the existing streaming engines (*i.e.*, [28, 29, 30]).

Stream Join Algorithms — An early stream join was formalized by Kang's three-step procedure [21]. Subsequently, Gedik *et al.* [6] introduced the parallel CellJoin, designed for a heterogeneous architecture, aiming to substantially improve stream join processing performance. However, CellJoin requires a re-partitioning task for each newly incoming tuple, which limits its scalability [6]. The problem of distributed stream join processing has also been studied with respect to elasticity and reduction of memory footprint, applicable to cloud computing [22].

Teubner *et al.* introduced a bi-direction data flow-oriented stream join processing approach, called the handshake join [9]. To reduce delay in the linear chaining, Teubner *et al.* [10] introduced a low-latency handshake join that uses a fast forwarding mechanism to expedite tuple delivery to all sub-windows by replicating every tuple k times, where the stream is split over k join cores. This mechanism is illustrated in Figure 10. Furthermore, the bi-directional flow complicates the logic for serializing the two pipes connecting consecutive join cores that is necessary in order to avoid race conditions due to concurrent in-flight tuples (*i.e.*, tuples traveling between neighboring processing cores).

Stream Processing Acceleration — Stream processing has received much attention over the past few years. Many viable research prototypes and products have been

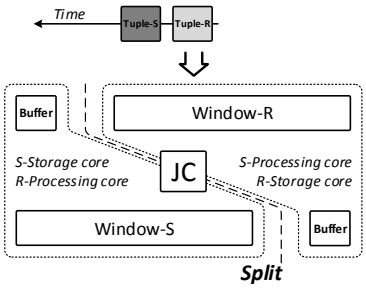


Figure 4: SplitJoin concept.

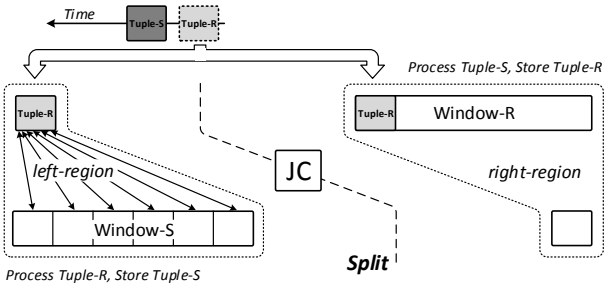


Figure 5: SplitJoin storing and processing steps.

developed, such as NiagaraCQ [24], TelegraphCQ [23], and Borealis [25], to just name a few. Most existing systems are fully software-based and support a rich query language, but stream join acceleration has not been the main focus of these approaches.

Since the inception of stream processing, the development of optimizations both at the query-level and at the engine-level have been widely explored. For example, co-processor-based solutions utilizing GPUs [8, 6] and more recently hardware-based solutions employing FPGAs have received attention [14, 15, 17, 18, 31]. For example, Tumeo *et al.* demonstrated how to use GPUs to accelerate regular expression-based stream processing language constructs [8]. The challenge in utilizing GPUs lies in transforming a given algorithm to use the highly parallel GPU architecture that has primarily been designed to perform high-throughput matrix computations and not, foremost, low latency processing.

Past work showed that FPGAs are a viable option for accelerating certain data management tasks in general and stream processing in particular [14, 15, 17, 18, 31, 32, 33]. For example, Hagiescu *et al.* [15] identify compute-intensive nodes in the query plan of a streaming computation. To increase performance in the hardware design that realizes the streaming computation, these nodes are replicated, which, due to the stateless nature of the query language considered, poses few issues. A main difference from our work is the restriction to stateless operations and the lack of a capability to flexibly update the streaming computation. Similarly, Mueller *et al.* [17] present Glacier, a component library and compiler, that compiles streaming queries into logic circuits on an operator-level basis. Both approaches are characterized by the goal of hardware-aware acceleration of streams, yet our solution is also applicable to non-FPGA parallel hardware.

4 SplitJoin

In this section, we describe SplitJoin and highlight two of its key properties, namely, the top-down data flow and the splitting of the join computation into independent storage and processing steps. Together, these properties remove any need for coordination and dependencies among join cores, which enables a high-degree of parallelism for SplitJoin without sacrificing latency.

4.1 SplitJoin Overview

SplitJoin diverts from the bi-directional data flow-oriented processing of existing approaches [9, 10]. As illustrated in Figure 1, SplitJoin introduces a single top-down data flow that fundamentally changes the overall tuple processing architecture. First, the join cores are no longer chained linearly (*i.e.*, avoiding linear latency overhead). In fact, they are now completely independent (*i.e.*, also avoiding inter-core communication overhead). Second, both streams travel through a single path entering each join core; thus, eliminating all complexity due to potential race conditions caused by in-flight tuples and complexity due to ensuring the correct tuple-arrival order, namely, the FIFO property is trivially satisfied by using a single (logical) path. Third, the communication path can be fully utilized to sustain the maximum throughput and each tuple no longer needs to pass every join core.

Another important aspect of SplitJoin is the simplification and decomposition of join processing itself. SplitJoin splits the dominant join abstraction that enforces the "storing" and "processing" steps to be coupled and done in a serial order. SplitJoin views these steps as two independent steps, namely, (i) "storing" and (ii) "processing". In fact, SplitJoin goes one step further and shows that not only these steps could be done in parallel, they can also be distributed to independent join cores. Therefore, unlike traditional parallel join processing that divides a single window into a set of sub-windows, where each is assigned to a core, SplitJoin introduces separate *storage* and *processing* cores that operate independently of each other as shown in Figure 4. The storage core is responsible for storing new tuples, while the processing core is responsible for the actual join operation of a new tuple in one stream with the existing tuples in the other stream.

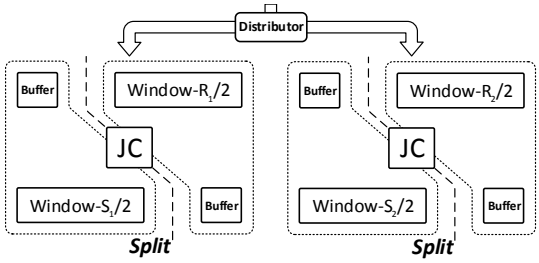


Figure 6: SplitJoin parallel architecture.

The splitting line in Figure 4 conceptually divides our join processing architecture into two separate parts, in which a *region* represents a stream's window and the associated buffer. We use the term *right-region* when referring to *Window-R* and *left-region* for *Window-S*. For each incoming tuple, a region either does processing or storing.

The split mechanism is illustrated in Figure 5, where the incoming tuples are fed to SplitJoin one after another. In the first step, `Tuple-R` is inserted into both regions. The right-region is responsible for storing `Tuple-R` in its sliding window, while the left-region is responsible for the processing of the replicated copy of `Tuple-R` (*i.e.*, the join comparison). The temporary tuple replication eliminates all inter-region communication among storage and processing cores. The replicated tuples are simply discarded once the processing is completed.

4.2 SplitJoin Parallelism

In SplitJoin, we parallelize the stream join computation by dividing each sliding window into a set of disjoint sub-windows. Each sub-window is assigned independently to a join core as shown in Figure 6 (*i.e.*, acting as a local buffer for each core). Each join core (*JC*) consists of a *left-* and a *right-region*. The division of the sliding window among join cores is accompanied by a *Distributor* unit to transmit incoming tuples to the join cores.

In the parallelized version of SplitJoin, all join cores receive the new incoming tuple. In each join core, depending on the tuple origin *i.e.*, whether *R* or *S* stream, the processing and storage steps are orchestrated. For example, if the incoming tuple belongs to the *R* stream, `Tuple-R`, then all processing cores dedicated to the left-region compare `Tuple-R` against all the tuples in the *S* stream sub-windows. Simultaneously, `Tuple-R` is also stored in the storage core of exactly one right-region. The assignment of `Tuple-R` follows an arbitration of the tuple to a storage core based on a round-robin selection. In other words, each region, based on its position number [5] and the number of seen tuples, independently determines its turn to store an incoming tuple. The proposed assignment model eliminates the need for a central coordinator for tuple assignment, which is a key contributor for achieving scalability in SplitJoin architecture. Notably, transmitting an incoming tuple to each join core translates into writing a tuple to the join core's local buffer (independent of any other join cores) that resembles a simple queue with a single producer and a single consumer, in which the producer is the *Distributor* and the consumer is join core itself.

4.3 Scalable Distribution Tree

The decoupling of storage and processing in SplitJoin simplifies parallelization by distributing sub-windows among many independent join cores. To fully leverage potential parallelism, we also need an efficient tuple distribution and routing mechanism.

In SplitJoin, to distribute the stream's transmission load in a balanced and scalable manner, we use a *k-ary* tree as the distribution network. As the network grows in size, the *Distributor* is replicated and its replicas are placed in the tree's inner nodes to achieve the desired scalability. As the number of SplitJoin join cores increases, we increase the fanout of each *Distributor* before increasing the depth of the distribution tree.

By applying replication recursively, we scale the distribution network as well as the number of join cores for SplitJoin. The resulting system, including the input data distribution network, SplitJoin's join cores, and the output data gathering network (similar in structure to the input network), is shown in Figure 11, where the horizontal bars illustrate the input distribution and output gathering networks.

The distribution network is the same for both count-based and time-based sliding window joins. However, in the time-based version each tuple carries an extra field for its timestamp. This field is to keep track of the lifespan of each tuple to realize the time-based sliding window semantic.

4.4 Expiration & Replacement Policies

Tuple expiration is a crucial step to ensure the correctness of the stream join semantic. In the count-based sliding window, the number of tuples in each window is specified explicitly while in the time-based sliding window a lifespan l (*e.g.*, $l = 10$ minutes) defines when a tuple must be expired.

SplitJoin supports both passive and active expiration techniques. The passive approach is primarily intended for the count-based sliding window, in which the incoming tuples simply overwrite the oldest tuples in the window. The expiration is done implicitly and mimics the functionality of a FIFO buffer. Once a window is full, the stored tuples are expired in order of their arrival. In the active expiration, geared towards the time-based sliding window, each join core locally manages the expiration of tuples from its sub-window. The expiration task for each sub-window is postponed until a tuple from the opposing stream with timestamp of t is received for processing. Then, in the region responsible for processing, just prior to the join computation, for each tuple with a timestamp t_i, if $(t - t_i) > l$, then the tuple is expired. Basically, tuples are expired when they fall off the user-defined lifespan (l) of the time-based window size.

Note the expiration is a local operation within a region and does not involve global coordination because tuples arrive with monotonically increasing timestamps and order is preserved when they are added and stored in a sub-window. The expiration task starts from the end (with the oldest tuple) of each sub-window and ends when a tuple younger than the user-defined lifespan is found. In other words, instead of sending explicit expiry messages with a timestamp, we rely on the timestamp of tuples in the input streams that must

[5] Position number refers to the logical location of a join core among other join cores.

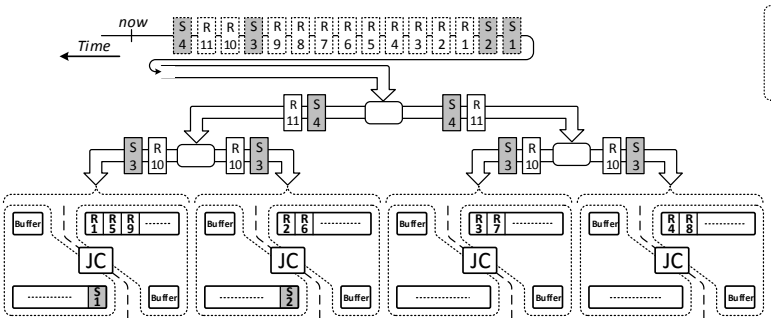


Figure 7: SplitJoin data distribution and processing example.

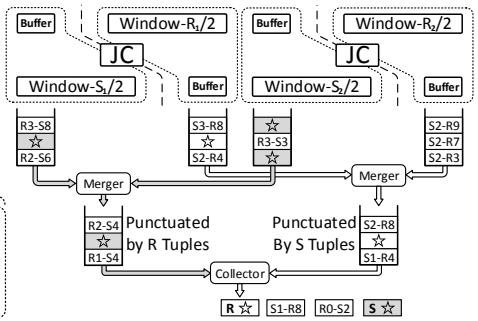


Figure 8: Punctuations in result gathering.

Algorithm 1: SplitJoin distribution network.

```
1 SplitJoin() begin
2     while still a tuple to consume do
3         broadcast tuple t to all Join Cores begin
4             forall join cores do
5                 Join_Core(t, source);
```

be routed to all nodes anyway. Therefore, the expiration messages are implicitly piggybacked on the incoming tuples as a way to broadcast the synchronized time without the need for global coordination.

In Figure 7, we illustrate how tuples are stored and processed in SplitJoin join cores. Assuming that we have a sequence of tuples as shown in the upper part of the figure, each tuple is transferred by the distribution tree to all join cores. Each `Tuple-R` is stored in exactly one *right-region* while processed by the *left-region* of all join cores. Likewise, for tuples from the *S* stream, they are stored in the *left-regions* and are processed by the *right-regions*.

As we can see in Figure 7, the tuples are distributed in-order. The tuples reach the storage cores through the same path (*i.e.*, the top-down flow), and the expiration procedure is preformed based on the order of incoming tuples (for both count-based and time-based sliding windows). Thus, unlike the bi-directional model used in [9, 10], neither the concurrency nor the race condition issues arise.

During processing, each region emits resulting tuples to be collected by the result gathering network (*cf.* Section 5). The processing step for each tuple in each region is completed by emitting an end notice from that region, referred to as a *star* punctuation mark. These marks serve to preserve the order of join results as we describe in detail in Section 5.

4.5 SplitJoin Algorithms

Tuple distribution in SplitJoin is specified in Algorithm 1. Upon arrival of a new tuple, regardless of its source stream, the tuple is broadcast to all join cores (Line 3).

In each join core, as presented in Algorithm 2, depending on the tuple's source (Lines 2 and 11), from *R* or *S* stream, the tuple is sent to the *right-region* for storage and to the *left-region* for processing or vice versa.

Finally, in the expiration process specified in Algorithm 4, the tuples that are too old to be considered for the join are expired from the end of the sub-window by computing their lifespan using their timestamp and the timestamp of the new tuple.

The pseudo code for the processing core (*i.e.*, the join comparison) is specified in Algorithm 3. An incoming tuple is compared with all tuples in the opposite sub-window (Lines 2-4). More importantly, this step is executed concurrently for each sub-window in every region. After processing (Lines 5-6), based on the chosen ordering precision, the *star* marker is produced and emitted. Also note that the goal of SplitJoin is to provide an efficient and coordination-free architecture for performing stream joins, and the particular choice of join algorithm is orthogonal. In this work, we adopted a simple variation of nested-loop join; however, within each core, one may choose any join algorithms such as hash- or index-based join.

5 Punctuated Result Collection

In SplitJoin, we employ a result gathering network (similar to our data distribution network) and a punctuation technique to preserve the ordering for the join result output. The full architecture of SplitJoin, that includes the distribution network, join cores (*JCs*), and the collection network, is illustrated in Figure 11.

In SplitJoin, we utilized a *2-ary* collection tree to gather and merge join results as depicted in Figure 8. The result

Algorithm 2: A join core in SplitJoin.

```
1  Join_Core(t, source) begin
2      if source = Stream R then                    // right-region
3          Expiration_Process(t, sub-window S);
4          Processing_Core(t, sub-window S);
5          if R_store_counter = node_id then
6              Storage_Core(t, sub-window R);
7          if R_store_counter = number of join cores then
8              R_store_counter ← 0;
9          else
10             R_store_counter ← R_store_counter + 1;
11     else                                         // left-region
```

Algorithm 3: Matches between **t** and sub-window **X**.

```
1 Processing_Core(t, sub-window X) begin
2     forall t_i-tuple in sub-window X do
3         compare t_i-tuple with t; if match then
4             emit the matched result;
5         if i≡0 (mod ordering_precision) then
6             emit punctuation star;
```

tuples of each processing core are gathered from the leaves of the collection tree. Each core has its own dedicated FIFO buffer. The collection tree employs a *Merger* unit and a FIFO buffer in each of its intermediate nodes (except in the root). Moving toward the tree's root (from top to bottom), at each node, the data in the two input buffers is merged into the buffer of that node. Merging continues up to the root, which contains the last buffer emitting the gathered join results.

5.1 Punctuation-based Ordering

SplitJoin architecture preserves the ordering of result tuples. The precision of the output order can be determined by a tunable system parameter, without significant changes in the processing architecture. To realize this flexibility in our design, we developed a *relaxed adjustable punctuation* (RAP) strategy. We define two levels of ordering guarantees for join results: the *outer* and *inner ordering*.

Definition 1 *The outer ordering of join results ensures that for any two consecutive incoming tuples, join results of the first tuple always precede the join results of the second tuple.*

Definition 2 *The inner ordering of join results ensures that for a single incoming tuple in one stream, join results are ordered in ascending order from the oldest to the most recently inserted tuple in the other stream.*

Our proposed relaxation enables us to maintain strict outer ordering while adjusting the precision of the inner ordering (essentially, not maintaining the inner ordering) in order to substantially reduce the overall cost of ordering. Furthermore, our technique supports strict outer and inner ordering as well.

In RAP, we define a simple punctuation emission rule for each core (the same simple rule applies to all cores), that is, the emission of a punctuation at the end of the processing of every newly inserted tuple (preserving the outer ordering and relaxing the inner ordering). In other words, each join core emits a punctuation after the end of processing a newly inserted tuple with all tuples in the other window. We

Algorithm 4: Expiring old tuples for time-based version.

```
1 Expiration_Process(t, sub-window X) begin
2     i← the end of sub-window X;
3     while t_i.timestamp - t.timestamp > Time Window Size do
4         omit t_i from sub-window X;
5         i←i−1;
```

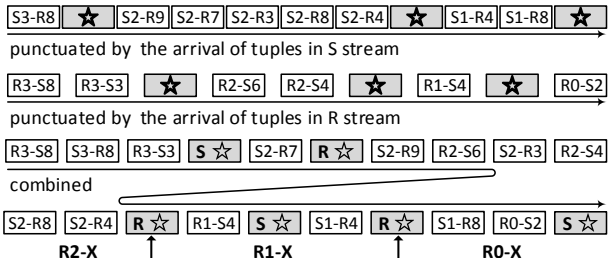


Figure 9: Punctuated resulting stream.

differentiate this punctuation from result tuples by a *star*, as shown in Figure 8.

SplitJoin cores insert both the join results and punctuation marks to collection tree leaves. The punctuation acts as a border between the join results of two consecutively inserted tuples (outer order). As join results and punctuations are pushed down the collection tree towards the root, at each node of the tree, the join results and their corresponding punctuation marker (stars) from the two buffers are merged into the FIFO buffer of their parent node. When the *Merger* in the parent node receives a star from one of its inputs, it disables that input and continues to receive resulting tuples from the other buffer until it receives a star from that buffer as well. The *Merger* merges two punctuations (stars) into one and pushes it to its FIFO buffer. This scenario repeats until the star reaches the output of the collection tree.

Since join results are pushed down in the order in which the newly inserted tuple arrives, the outer ordering for each core is trivially satisfied due to the single top-down FIFO flow of SplitJoin that starts from the root of the distribution tree (for inserting new tuples) and ends at the root of the collection tree (for merging the join results). This flow is shown in Figure 9.

The final step in the result gathering network employs a *Combiner* rather than a *Merger*. On the right side of the split are the punctuated results, ordered by the tuples from the *S* stream, while on the left side, the punctuation is based on the arrival sequences of tuples from the *R* stream. These two sets of punctuated result tuples are consumable as separate streams. However, to emit only one stream as output, we use a *Combiner* which simply fetches the resulting tuples and punctuations from their input and puts them into the output buffer. The *Combiner* keeps track of the origin of punctuations (whether from the right- or left-regions) by flagging the *stars* with *R* and *S*, as shown in Figure 9.

In Figure 9, the upper flow is the result stream from the right-regions, punctuated by the order of *S* stream tuples, while the middle one is the result stream from the left-regions, punctuated by the order of *R* stream tuples. The lower flow demonstrates the combined result stream that includes all result tuples in addition to punctuations. For example, **R1-X** is specified by two *R* punctuations and includes the result tuples which start with **R1**.

Adjusting the punctuation interval is straightforward and

Algorithm 5: Punctuation-based N-ary merger.

```
1 N-ary_Merger(t) begin
2     foreach right(or left)-region of core_1..N in sequence do
3         while a resulting tuple (t) is available in output buffer till
            the first star do
4             pop t from join core's output buffer;
5             push t to Merger's output buffer;
6     push out the end of result star;
```

only requires us to tune the punctuation emission rate in SplitJoin's cores. Each core can simply change the frequency at which a punctuation is generated. For example, each core can be tuned to produce a punctuation after joining one newly inserted tuple (strict outer ordering) or after every five tuples (relaxed outer ordering). We could also adjust the precision of inner ordering by increasing the frequency of punctuation generation. For example, to produce a strict inner ordering, each incoming tuple is compared with tuples in the opposite window (starting from the oldest to the most recently inserted one), followed by outputs for both the join result and the punctuation marker for every comparison. Therefore, if each core has a window size of w, then up to w punctuation markers (*i.e.*, *stars*) are produced for every newly inserted tuple. For a relaxed inner ordering, only one punctuation is produced after joining the incoming tuples with all the tuples in the opposite window. At the other extreme, when no ordering is required, we could simply disable the punctuation generation altogether.

5.2 Ordering Algorithm

For the result gathering network, we utilized a *k-ary* tree. Algorithm 5 specifies the pseudo-code for an *N-ary* (*e.g.*, 2-ary) *Merger* given an *N-ary* result gathering tree. The *Merger* is connected to the output FIFO buffer of N regions and collects the resulting tuples and punctuations into its own output FIFO buffer, which is subsequently fed to the next intermediate node (its parent) in the tree. This is repeated up to the root of the tree, where result tuples are punctuated by tuple arrival order from the two stream types (either R or S).

The *Merger* connects to the output buffers of the same source, either *left* or *right* regions (*cf.* Line 2 of Algorithm 5). Each *Merger* collects the results in the same order as the join cores store the new incoming tuples. For example, assuming that the first `Tuple-R` is stored in the left most join core in its *right-region*, as shown in Figure 7. The *Merger* then begins the collection of results from the comparison of `Tuple-S` with the R sub-window in the left most *right-region* as well.

In the result gathering, the *Merger* fetches tuples from the first region's buffer and stores them in its own output buffer until it reaches the first *star* in (Line 3∼Line 5). Then it repeats the same procedure for the next region's buffer until it receives a *star* from there too.

After receiving a punctuation mark from the last region, the *Merger* forwards the punctuation to its output buffer (*cf.* Line 6). Note that each *Merger* emits only one punctuation mark for every pair of punctuation (*i.e.*, one punctuation mark from each join core).

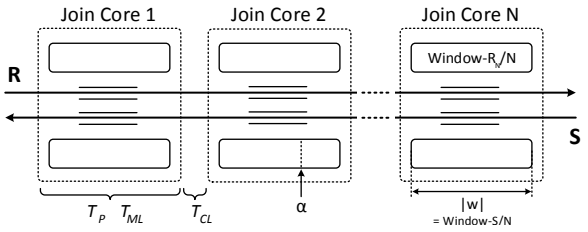


Figure 10: Low-latency handshake join overview [10].

Using a higher ordering precision increases the number of punctuation marks between result tuples of each region. For example, instead of having one punctuation mark after comparison of a tuple with the whole sub-window, we can have one punctuation after each 10 comparisons. Since tuples in the sub-window are already stored in the order of their arrival, the intermediate punctuations preserve the result ordering while gathering the results from all the join cores. For obtaining a higher precisions, *Mergers* follow the same procedure as before.

6 Runtime Complexity

In this section, we present a brief analytical model to study the runtime complexity of SplitJoin relative to related techniques [9, 10]. In the analysis, we use the following definitions.

Definition 3 *We define the processing latency (PL) as the time from when a tuple arrives at the join operator until the tuple is compared and joined with all tuples in the other window and all the matching results are produced.*

Definition 4 *We define the visiting latency (VL) as the time required for two tuples from both streams to be compared with each other.*

6.1 Low-latency Handshake Join Analysis

The processing latency for low-latency handshake join (*cf.* Figure 10) is given as follows:

$$PL = T_{CL} + ((k-1) \times (T_{CL} + T_{ML})) + (w \times T_P) + T_{Col} \quad (1)$$

where T_{CL} represents the communication time between cores and k the number of processing cores. T_{ML} accounts for the tuple monitoring time in both streams, required to prevent missing results between tuples in the fast-forwarding buffers (i.e., race conditions). Therefore, the cost of propagating a single tuple to all cores by replication and fast-forwarding is captured by $((k-1) \times (T_{CL} + T_{ML}))$. The size of each sub-window in each core is denoted by w. T_P represents the processing time to perform the join operation between each pair of tuples. To simplify the analysis, we assume that all join cores are working in parallel. Finally, T_{Col} presents the

time required to collect all the matching results. In [10] the authors rely on a linear collector method for gathering the results that has the potential to break the strict neighbor-to-neighbor communication model of handshake join.

In theory, assuming a fixed-size sub-window, we can increase the number of processing cores to support larger windows. Therefore, the join's latency scales linearly in the number of processing cores, *i.e.*, as $O(k)$, — optimistically assuming that central coordination would not become a bottleneck while ignoring the effect of the result collection method.

To calculate the visiting latency, we assume that the number of in-flight tuples from the two streams that must be compared (*i.e.*, the monitoring time T_{ML}) is negligible. While this assumption renders the model less realistic (which was also implicitly assumed in [10]), it simplifies the visiting latency analysis.

Any pair of tuples from both streams meet each other in, at most, one location; let this location be α as shown in Figure 10. α could be in any core. If α happens to be on the first core, then the latency is lower, while if it is on the last core, then the latency is higher. Thus, we define the average visiting latency as follows:

$$VL_{avg} = T_{CL} + (\lfloor \frac{(k-1)}{2} \rfloor \times (T_{CL} + T_{ML})) + ((\frac{w}{2}) \times T_P) \quad (2)$$

$(\lfloor \frac{(k-1)}{2} \rfloor \times (T_{CL} + T_{ML}))$ determines the average time to reach location α (essentially, reaching the mid-point of core chain) and $(\frac{w}{2}) \times T_P$ captures the processing time for half the tuples at α. The visiting latency scales linearly with the number of processing cores $O(k)$, assuming $(T_{CL} + T_{ML})$ is constant, irrespective of the number of cores.

To simplify the analysis, we ignore the overhead of central coordination in low-latency handshake join [10]. The coordinator requires sending an explicit expiry message for every tuple [10]. On average, these messages double the communication traffic between the central coordinator and each join core, significantly affecting the performance as observed in our experimental evaluation.

6.2 SplitJoin Analysis

SplitJoin utilizes a distribution tree to deliver incoming tuples to each join core in $O(log_b k)$ time, where k is the number of join cores and b is the branching factor of the distribution tree. We define $Path_{1\cdots k}$ as a distribution route that a tuple must travel to reach the join cores $1 \cdots k$, respectively. Let $T_{C_{Path_i, Depth_j}}$ be the communication cost (duration) of transferring a tuple to the i^{th} path at depth j. We define the processing latency for SplitJoin as follows:

$$PL = \max_{i=1\cdots k} \Big(\sum_{j=1\cdots log_b k} T_{C_{Path_i, Depth_j}} + (w \times T_{P_i}) \Big) + T_{Col} \quad (3)$$

where T_{P_i} is the processing time to perform the join operation between each pair of tuples for the i_{th} core. Assuming the communication times, $T_{C_{Path_i, Depth_j}}$, are roughly equal, then it follows that:

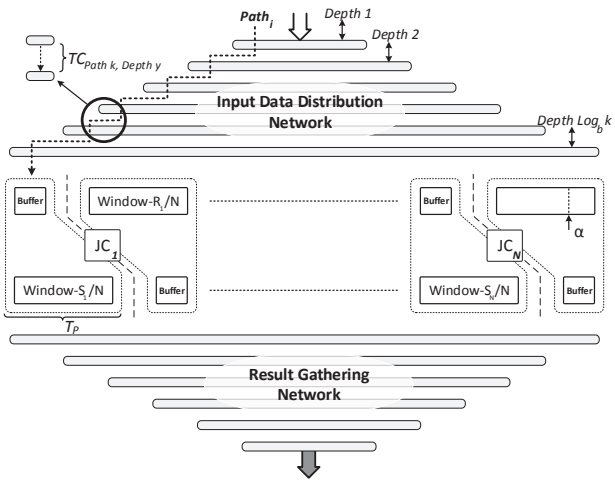


Figure 11: SplitJoin complete system.

$$PL = \max_{i=1\cdots k} (T_{C_{Path_i}} \times log_b k) + (w \times T_{P_i})) + T_{Col} \quad (4)$$

If we further assume homogeneous join cores and homogeneous distribution routes within the tree and also decompose T_{Col} into smaller units of work, then it follows that:

$$PL = (T_{CL} \times log_b k) + (w \times T_P) + (T_{CL} \times log_c k) \quad (5)$$

where $\log_c k$ defines the depth of the result gathering tree with the branching factor of c (from root to leaves). Assuming a fixed-size sub-window, as we increase the number of join cores, latency increases logarithmically, $O(\log_b k)$ (assuming $b < c$), for SplitJoin as opposed to the linear increase ($O(k)$) observed in [10].

Supposing two consecutive tuples from both streams meet at the point α, as shown in Figure 11, then their communication times in the distribution tree mostly overlap with each other because they are pushed to the distribution tree one after another. They travel together (using the FIFO strategy) to reach the targeted join core. As above, here, we also assume homogeneous join cores and communication costs within the distribution tree. Then, the average visiting latency of SplitJoin is given by:

$$VL_{avg} = (T_{CL} \times log_b k) + (\frac{w}{2} \times T_P) \quad (6)$$

Thus, the average visiting latency is also logarithmic in the number of join cores, compared to the linear order in [10].

7 Experimental Results

In this section, we experimentally evaluate our SplitJoin implementation. All experiments are performed on a 32-core system. Our system is a Dell PowerEdge R820 featuring 4 × Intel E5-4650 processors and 32 × 16GB DDR3 memory (RDIMM, 1600 MHz, Low Volt, Dual Rank, x4). We ran our benchmarks on Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-57-generic x86_64) installed on a Docker container [34] running on the same host OS.

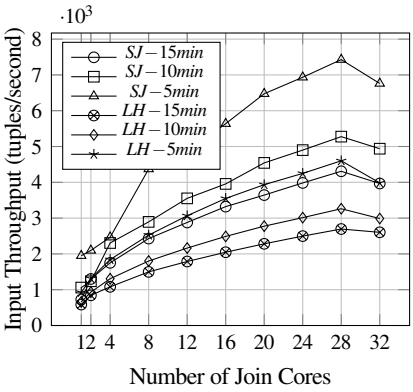


Figure 12: Throughput comparison.

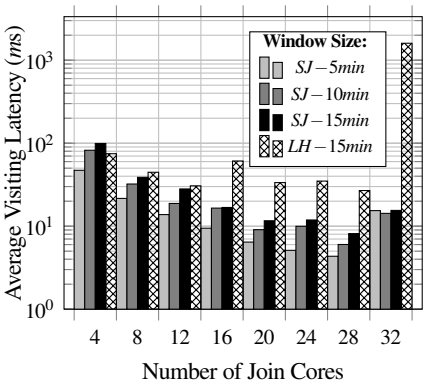


Figure 13: Visiting latency comparison.

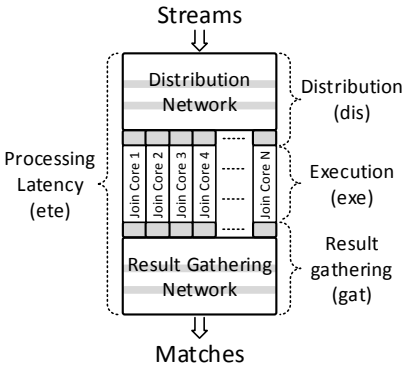


Figure 14: SplitJoin processing pipeline.

7.1 Experimental Setup

We adopted the benchmark used in recent stream join approaches [6, 9, 10]. In this benchmark two streams $R =$ `(x:int, y:float, z:char[20])` and $S =$ `(a:int, b:float, c:double, d:bool)` are joined via the two-dimensional band join, as follows:

```
WHERE r.x BETWEEN s.a-10 AND s.a+10
  AND r.y BETWEEN s.b-10 AND s.b+10
```

In our evaluations, we used the low-latency (referred to as *LH vs.* SplitJoin (*SJ*)) and the original handshake join libraries that were kindly provided by the authors of [9, 10]. Also in line with related approaches, integers and floats were generated following a uniform distribution in the range of $1-10^4$, unless otherwise stated.

The results cover the end-to-end evaluation, including data distribution network, SplitJoin storage and processing cores, the result gathering network, and also the proposed punctuated ordering mechanism. The punctuation precision is based on the outer tuple ordering as shown in Figure 9, unless otherwise stated.

In our time-based window realization, we generated timestamps on-the-fly using the system call `clock_gettime()`. Using a synthetic timing mechanism, as we experimented, further improves the overall performance by about 15% by relieving the overhead incurred by system calls.

7.2 Performance & Scalability

We evaluate SplitJoin performance by measuring latency and throughput metrics as we scale the level of parallelism. In general, key factors that influence the stream join performance are how the input streams are flowing through the join cores and how the joined results are collected and flow to the output.

In Figure 12, we demonstrate throughput results of SplitJoin in comparison with [10]. As we scale the number of join cores, we observe that both solutions scale gracefully; however, SplitJoin outperforms the low-latency handshake join by up to 60% (comparison between 15-min sliding windows). Theoretically, the performance of both approaches should be similar, as both utilize all join cores in parallel to process incoming tuples. However, the core-to-core communication and mandatory expiry messages in low-latency handshake join (necessary for both time-based and count-based join versions) impose a noticeable penalty.

In Figure 12, we also observe how the two approaches perform for different time-based window sizes. When the join core count is 32, we observe a drop in performance in both of the approaches. This is due to the existence of extra threads to perform other (non-processing) tasks such as stream distribution and result gathering in case of SplitJoin, and tuple assignment, expiry message generation, and result gathering in case of the low-latency handshake join. Since our system has only 32 processing cores, by instantiating 32 join cores, the operating system is forced to perform context switches, resulting in system saturation and performance drop.

7.3 Latency Evaluations

In Figure 14, we present an abstract model of our end-to-end processing pipeline stages. The grayed parts show intermediate and pipeline buffers. In the measurements, we are reporting the latency of the distribution stage (*dis*), which also includes the time that tuples are waiting in the pipeline stage between the distribution and execution stages. The latency of the execution stage (*exe*) is the latency attributed to the time that it takes a tuple to pass through the processing and storage steps in the join core, which also includes the tuple expiration process for the time-based sliding window. The latency for the last stage includes the time that resulting tuples are waiting in the pipeline stage between the execution and result gathering stages and also the time for the *Merger* and the *Collector* units to bring them to the output of SplitJoin. Latency reports for these measurements plus the processing, end-to-end (*ete*), and latency of SplitJoin, for the time-based sliding window, are presented in Figure 15.

Processing Pipeline Stage Latency: In the distribution network, as we increase the number of join cores, incoming tuples are distributed between larger number of join cores instead of having to pile up in the pipeline buffer for fewer join cores. Therefore, increasing the number of join cores, inherently reduces the waiting time in the distribution stage

as shown in Figure 15.

Since our evaluation system has only four processor sockets, the increase in the size of the distribution network has no significant effect on the performance except when the size of the sliding window is small. In the execution stage, the increase in the number of join cores for a given window size translates into smaller sub-windows for each join core and the latency also proportionally decreases.

Among our three pipeline stages, the result gathering network with punctuation ordering had the highest latency impact. This latency was mainly due to the waiting times of the *Mergers* on one of their input ports to receive a punctuation mark (*star*) before starting to read from their next port.

Visiting Latency: In Figure 13, we observe the average visiting latency ($T_{match} - max(t_r,t_s)$) for SplitJoin with 5, 10, and 15 minutes sliding windows, and low-latency handshake join with a 15 minutes sliding window for varying number of join cores. The t_r and t_s stand for initial timestamp of r and s tuples, respectively.

As we evaluated the average visiting latency (*cf.* Section 6), the latency increases logarithmically, $O(\log_b k)$, for SplitJoin as opposed to linearly, $O(k)$, for the low-latency handshake join [10]. By comparing the average visiting latency for the 15-min version of SplitJoin and low-latency handshake join, when we use four join cores, the latency is quite similar; however, once the number of join cores increases, the gap between SplitJoin and low-latency handshake join widens drastically by a factor of up to 3.3X (8.1*ms vs.* 26.8*ms* for 28 join cores).

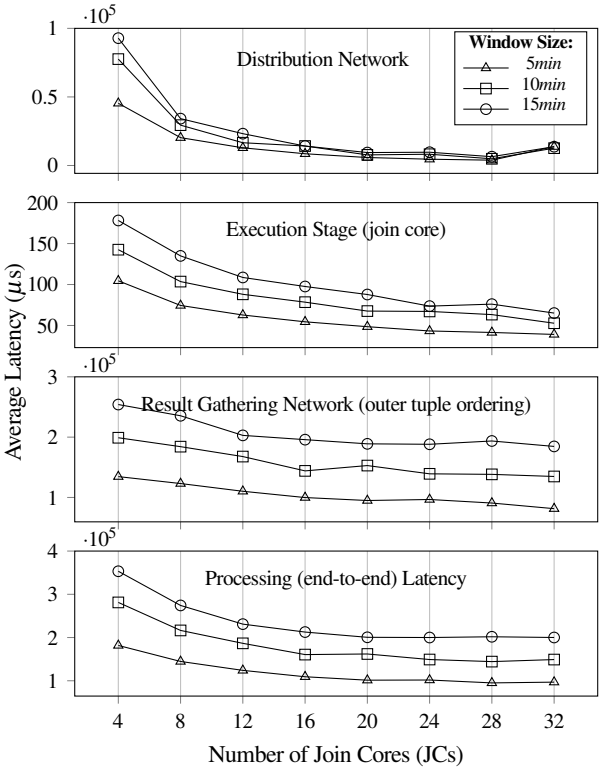


Figure 15: SplitJoin latency measurements.

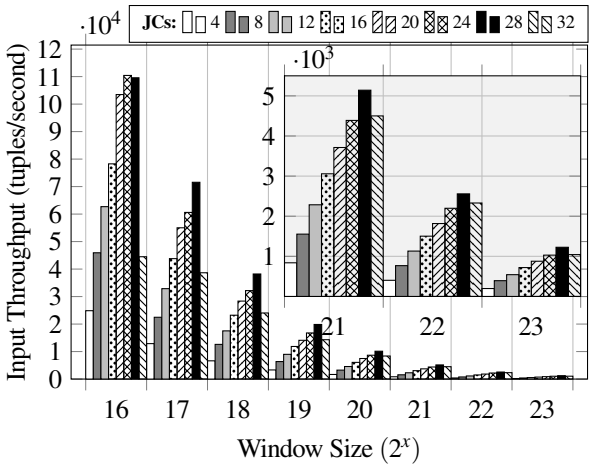


Figure 16: Count-based SplitJoin throughput.

We observe an increase in latency while reaching 32 join cores which is again due to the lack of enough resources for the other (non-processing) tasks. Since low-latency handshake join requires to perform additional costly tasks, such as emitting individual expiry message for each tuple, the resource contention shows a more significant impact on latency as seen when instantiating 32 join cores.

7.4 Count-based Sliding Window

Although the count-based and time-based versions of SplitJoin behave similarly, there are two key differences: (1) having no space allocated for timestamp values and no on-the-fly generation of timestamps through a costly system call and (2) having a fixed window size for count-based semantics as opposed to the time-based semantics where the window size varies depending on the incoming tuple rate. These differences result in roughly 20% improvement in performance for SplitJoin using a count-based instead of time-based sliding window. For example, SplitJoin instantiated with 28 join cores over a 15-min sliding window (shown in Figure 12) sustains an input rate of 4400 tuples/second, which roughly translates to window sizes of 2^{21} for each stream. But for the count-based window, if we set the window size to (2^{21}), SplitJoin can process up to 5200 tuples/second, as shown in Figure 16.

In the count-based results shown in Figure 16, we observe two effects: (1) larger window sizes result in fewer punctuation marks, assuming the same input throughput, since in the outer tuple ordering each join core produces one punctuation mark at the end of each tuple processing and (2) a larger sub-window per join core additionally increases the processing efficiency by reducing the impact of other (non-processing) pipeline stages. Based on these observations, doubling the window size while fixing the number of join cores reduces the processing throughput.

For each window size, by increasing the number of join cores (**JCs**), we observe a relative improvement in the

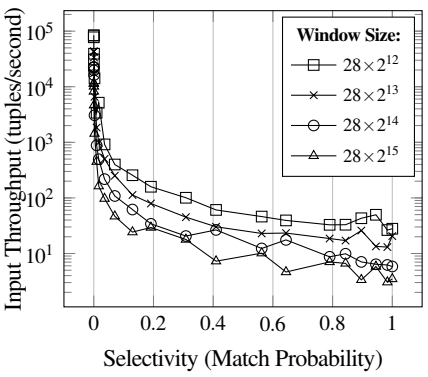


Figure 17: Selectivity effect on SplitJoin throughput (28 JCs).

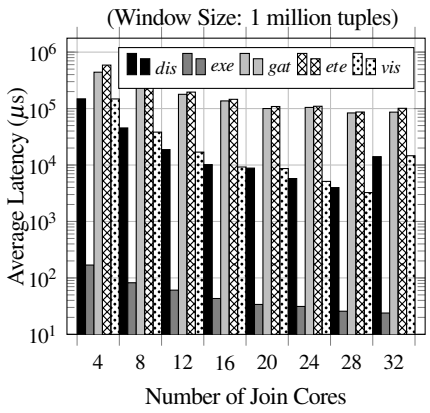


Figure 18: Count-based SplitJoin latency reports (uniform distribution:$1-10^5$).

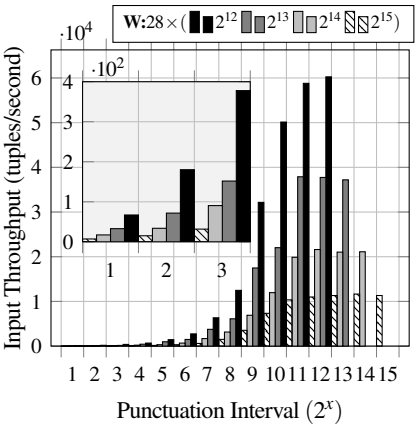


Figure 19: Ordering precision effect (28 JCs, uniform distribution:$1-10^4$).

throughput except when using 32 join cores. Over-utilizing system resources (*i.e.*, using 32 join cores) has more impact on the throughput for smaller window sizes. Larger windows keep join cores busier, thus, new tuples are processed after longer waits. This relieves other tasks (*i.e.*, distribution), reducing the effect of resource contention.

In Figure 18, we present the latency of the processing pipeline stages, the average processing latency (*ete*), and visiting (*vis*) latency for SplitJoin for the count-based sliding window. SplitJoin scales gracefully as we increase the number of join cores; in particular, using 28 join cores, the visiting latency is improved by more than 2.5X and 8.3X as compared to the time-based version of SplitJoin and the low-latency handshake join, respectively.

7.5 Effect of Selectivity

The selectivity (also called match probability) is one of the major factors affecting join performance. Often a low selectivity is assumed in most related work [6, 9, 10]. However, it is important to analyze the sensitivity of a join algorithm with respect to the selectivity in order to assess the generality of the approach.

The effect of varying the selectivity on the input throughput is illustrated in Figure 17. The key observation is that SplitJoin's latency scales reasonably, and it is robust to changes of selectivity, even for sliding windows as large as 28×2^{15} tuples.

7.6 Effect of Punctuation Precision

Figure 19 demonstrates the effect of the ordering precision on the processing performance. In this diagram, we utilize 28 join cores with varying sub-window sizes ($2^{12}-2^{15}$) per join core. The ordering precision starts from one punctuation per sub-window processing, referred to as *relaxed inner ordering*, and progressively increases the precision until one punctuation mark (*star*) is produced after each comparison (represented as 2^1 on the x-axis) within each sub-window, referred to as *strict inner ordering*.

The *relaxed inner ordering* is the same as the *strict outer ordering*. Therefore, the highest punctuation interval for each window size in Figure 19 represents the effect of *strict outer ordering* on the throughput for that window size.

As we increase the precision (*e.g.*, focusing on a sub-window size of 2^{15}), from $2^{11}-2^{15}$, its effect on the overall performance is negligible, since the number of punctuations produced per each incoming tuple in each join core is relatively low (*i.e.*, 1, 2, 4, 8, and 16 punctuations, respectively) compared to the sub-window size which is 2^{15}. However, as we continue to increase the precision from the interval 2^{10} down to 2^1, the number of punctuations becomes comparable to the sub-window size for each join core, and as expected, negatively affects the performance of SplitJoin. This highlights the importance of balancing ordering precision versus overall performance. In fact, since the precision is adjustable, to achieve a desired throughput, SplitJoin could adaptively adjust the precision interval to achieve a sweet spot between the ordering precision and the sustainable input throughput.

8 Conclusions

We present SplitJoin, a novel stream join that introduces two unique properties that distinguish it from existing work. First, SplitJoin exhibits a scalable architecture that splits the join computation into two independent "storing" and "processing" steps that can be parallelized using a coordination-free protocol to achieve low-latency join processing. Second, SplitJoin introduces a simplified top-down, flow-oriented join processing that eliminates complex concurrency logic for avoiding race conditions while satisfying input stream ordering semantics. We further propose scalable distribution- and collection-trees for input stream propagation and output join result gathering, respectively. Lastly, we propose a relaxed adjustable punctuation to guarantee join result ordering and to provide an effective mechanism to balance the trade-offs between the ordering precision and the overall join throughput.

Acknowledgment

This research was supported by the Alexander von Humboldt Foundation.

References

[1] SRIVASTAVA, D., GOLAB, L., GREER, R., JOHNSON, T., SEIDEL, J., SHKAPENYUK, V., SPATSCHECK, O., AND YATES, J. Enabling real time data analysis. VLDB'10.

[2] SADOGHI, M., JACOBSEN, H.-A., LABRECQUE, M., SHUM, W., AND SINGH, H. Efficient event processing through reconfigurable hardware for algorithmic trading. VLDB'10.

[3] CRANOR, C., JOHNSON, T., AND SPATASCHEK, O. Gigascope: a stream database for network applications. SIGMOD'03.

[4] FONTOURA, M., SADANANDAN, S., SHANMUGASUNDARAM, J., VASSILVITSKI, S., VEE, E., VENKATESAN, S., AND ZIEN, J. Efficiently evaluating complex Boolean expressions. SIGMOD'10.

[5] GEDIK, B., BORDAWEKAR, R. R., AND YU, P. S. CellSort: high performance sorting on the cell processor. VLDB'07.

[6] GEDIK, B., BORDAWEKAR, R. R., AND YU, P. S. CellJoin: A parallel stream join operator for the cell processor. VLDBJ'09.

[7] MARGARA, A., AND CUGOLA, G. High performance content-based matching using GPUs. DEBS'11.

[8] TUMEO, A., VILLA, O., AND SCIUTO, D. Efficient pattern matching on GPUs for intrusion detection systems. CF'10.

[9] TEUBNER, J., AND MUELLER, R. How soccer players would do stream joins. SIGMOD'11.

[10] ROY, P., TEUBNER, J., AND GEMULLA, R. Low-latency handshake join. VLDB'14.

[11] BLANAS, S., LI, Y., AND PATEL, J. M. Design and evaluation of main memory hash join algorithms for multi-core CPUs. SIGMOD'11.

[12] TEUBNER, J., ALONSO, G., BALKESEN, C., AND OZSU, M. T. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. ICDE'13.

[13] LEIS, V., BONCZ, P., KEMPER, A., AND NEUMANN, T. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. SIGMOD'14.

[14] MUELLER, R., TEUBNER, J., AND ALONSO, G. Data processing on FPGAs. VLDB'09.

[15] HAGIESCU, A., WONG, W.-F., BACON, D., AND RABBAH, R. A computing origami: Folding streams in FPGAs. DAC'09.

[16] NAJAFI, M., SADOGHI, M., AND JACOBSEN, H.-A. Flexible query processor on FPGAs. VLDB'13.

[17] MUELLER, R., TEUBNER, J., AND ALONSO, G. Streams on wires: a query compiler for FPGAs. VLDB'09.

[18] WOODS, L., TEUBNER, J., AND ALONSO, G. Complex event detection at wire speed with FPGAs. VLDB'10.

[19] SADOGHI, M., JAVED, R., TARAFDAR, N., SINGH, H., PALANIAPPAN, R., AND JACOBSEN, H.-A. Multi-query stream processing on FPGAs. ICDE'12.

[20] NAJAFI, M., SADOGHI, M., AND JACOBSEN, H.-A. Configurable hardware-based streaming architecture using online programmable-blocks. ICDE'15.

[21] KANG, J., NAUGHTON, J., AND VIGLAS, S. Evaluating window joins over unbounded streams. ICDE'03.

[22] LIN, Q., OOI, B. C., WANG, Z., AND YU, C. Scalable distributed stream join processing. SIGMOD'15.

[23] CHANDRASEKARAN, S., ET AL. TelegraphCQ: Continuous dataflow processing for an uncertain world. CIDR'03.

[24] CHEN, J., DEWITT, D. J., TIAN, F., AND WANG, Y. NiagaraCQ: A scalable continuous query system for internet databases. SIGMOD'00.

[25] ABADI, D. J., ET AL. The design of the Borealis stream processing engine. CIDR'05.

[26] WU, L., BARKER, R. J., KIM, M. A., AND ROSS, K. A. Navigating big data with high-throughput, energy-efficient data partitioning. ISCA'13.

[27] XIE, J., AND YANG, J. A survey of join processing in data streams. In *Data Streams*, C. Aggarwal, Ed., vol. 31 of *Advances in Database Systems*. Springer US, 2007.

[28] AKIDAU, T., BRADSHAW, R., CHAMBERS, C., CHERNYAK, S., FERNNDEZ-MOCTEZUMA, R. J., LAX, R., MCVEETY, S., MILLS, D., PERRY, F., SCHMIDT, E., AND WHITTLE, S. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. VLDB'15.

[29] KULKARNI, S., BHAGAT, N., FU, M., KEDIGEHALLI, V., KELLOGG, C., MITTAL, S., PATEL, J. M., RAMASAMY, K., TANEJA, S. Twitter Heron: Stream processing at scale. SIGMOD'15.

[30] Apache Storm. http://storm.apache.org.

[31] SOURDIS, I., AND PNEVMATIKATOS, D. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion. FPL'03.

[32] BORDAWEKAR, R. R., AND SADOGHI, M. Accelerating database workloads by software-hardware-system co-design. ICDE'16.

[33] NAJAFI, M., SADOGHI, M., AND JACOBSEN, H.-A. The FQP vision: Flexible query processing on a reconfigurable computing fabric. SIGMOD'15.

[34] MERKEL, D. Docker: Lightweight linux containers for consistent development and deployment. Linux J'14.

Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing

Keval Vora
University of California, Riverside
kvora001@cs.ucr.edu

Guoqing Xu
University of California, Irvine
guoqingx@ics.uci.edu

Rajiv Gupta
University of California, Riverside
gupta@cs.ucr.edu

Abstract

Single-PC, disk-based processing of big graphs has recently gained much popularity. At the core of an efficient disk-based system is a well-designed partition structure that can minimize random disk accesses. All existing systems use *static partitions* that are created before processing starts. These partitions have static layouts and are loaded entirely into memory in every single iteration even though much of the edge data is not changed across many iterations, causing these unchanged edges to have *zero new impact* on the computation of vertex values.

This work provides a general optimization that removes this I/O inefficiency by employing *dynamic partitions* whose layouts are dynamically adjustable. Our implementation of this optimization in GraphChi — a representative out-of-core vertex-centric graph system — yielded speedups of 1.5—2.8× on six large graphs. Our idea is generally applicable to other systems as well.

1 Introduction

As graphs have become increasingly important in modern computing, developing systems to efficiently process large graphs has been a popular topic in the past few years. While a distributed system is a natural choice for analyzing large amounts of graph data, a recent trend initiated by GraphChi [14] advocates developing *out-of-core* support to process large graphs on a single commodity PC.

Out-of-core graph systems can be classified into two major categories based on their computation styles: *vertex-centric* and *edge-centric*. Vertex-centric computation, that originates from the "think like a vertex" model of Pregel [17], provides an intuitive programming interface. It has been used in most graph systems (*e.g.*, [8, 16, 14, 10]).

Recent efforts (*e.g.*, X-Stream [23] and GridGraph [31]) develop edge-centric computation (or a hybrid processing model) that streams edges into memory to perform vertex updates, exploiting locality for edges at the cost of random accesses to vertices. Since a graph often has many more edges than vertices, an edge-centric system improves performance by reducing random accesses to edges.

Observation At the heart of both types of systems is a well-designed, disk-based partition structure, along with an efficient *iterative, out-of-core* algorithm that accesses the partition structure to load and process a small portion of the graph at a time and write updates back to disk before proceeding to the next portion. As an example, GraphChi uses a *shard* data structure to represent a graph partition: the graph is split into multiple shards before processing; each shard contains edges whose target vertices belong to the same logical interval. X-Stream [23] partitions vertices into *streaming partitions*. GridGraph [31] constructs 2-dimensional edge blocks to minimize I/O.

Despite much effort to exploit locality in the partition design, existing systems use *static partition layouts*, which are determined before graph processing starts. In every single computational iteration, each partition is loaded entirely into memory, although a large number of edges in the partition are not strictly needed.

Consider an iteration in which the values for only a small subset of vertices are changed. Such iterations are very common when the computation is closer to convergence and values for many vertices have already stabilized. For vertices that are not updated, their values do not need to be pushed along their outgoing edges. Hence, the values associated with these edges remain the same. The processing of such edges (*e.g.*, loading them and reading their values) would be completely redundant in the next iteration because they make zero *new contribution* to the values of their respective target vertices.

Repeatedly loading these edges creates significant I/O inefficiencies, which impacts the overall graph processing performance. This is because data loading often takes a major portion of the graph processing time. As an example, over 50% of the execution time for PageRank is spent on partition loading, and this percentage increases further with the size of the input graph (*cf.* §2).

However, none of the existing out-of-core systems can eliminate the loading of such edges. Both GraphChi and X-Stream support *vertex scheduling*, in which vertices are scheduled to be processed for the next iteration if they have at least one incoming edge whose value is changed in the current iteration. While this approach reduces unnecessary computations, it cannot address the I/O inefficiency: although the value computation for certain vertices can be avoided, shards still need to be entirely loaded to give the system accesses to all vertices and edges. Similarly, GridGraph's 2-D partition structure remains static

throughout computation regardless of the dynamic behavior of the algorithm — a partition has to be loaded entirely even if only one vertex in it needs to be computed.

Contributions This paper aims to reduce the above I/O inefficiency in out-of-core graph systems by exploring the idea of *dynamic partitions* that are created by omitting the edges that are not updated.

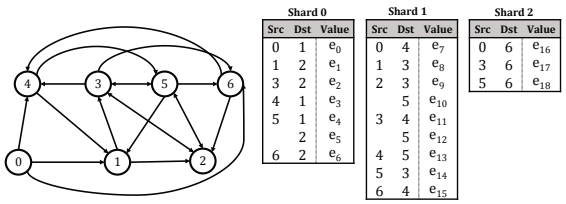
While our idea is applicable to *all* disk-based systems, in this work we focus on dynamically adjusting the *shard structure* used in GraphChi. We choose GraphChi as the starting point because: (1) it is a representative of extensively-used vertex-centric computation; (2) it is under active support and there are a large number of graph programs already implemented in it; and (3) its key algorithm has been incorporated into *GraphLab Create* [1], a commercial product of Dato, which performs both distributed and out-of-core processing. Hence, the goal of this paper is *not* to produce a brand new system that is faster than all existing graph systems, but instead, to show the *generality and effectiveness* of our optimization, which can be implemented in other systems as well.

Challenges Using dynamic partitions requires much more than recognizing unnecessary edges and removing them. There are two main technical challenges that need to be overcome.

The **first challenge** is how to perform vertex computation in the presence of missing edges that are eliminated during the creation of a dynamic partition. Although these edges make no impact on the forward computation, current programming/execution models all assume the presence of all edges of a vertex to perform value updates. To solve the problem, we begin with proposing a delay-based computation model (*cf.* §3) that *delays* the computation of a vertex with a missing edge until a special *shadow iteration* in which all edges are brought into memory from static partitions.

Since delays introduce overhead, to reduce delays, we further propose an *accumulation-based* programming/execution model (*cf.* §4) that enables *incremental vertex computation* by expressing computation in terms of contribution increments flowing through edges. As a result, vertices that *only have missing incoming edges* can be processed instantly without needing to be delayed because the increments from missing incoming edges are guaranteed to be zero. Computation for vertices with missing outgoing edges will still be delayed, but the number of such vertices is often very small.

The **second challenge** is how to efficiently build partitions on the fly. Changing partitions during processing incurs runtime overhead; doing so frequently would potentially make overheads outweigh benefits. We propose an additional optimization (*cf.* §5) that constructs dynamic partitions only during shadow iterations. We show, theoretically (*cf.* §5) and empirically (*cf.* §6), that this optimization leads to I/O reductions rather than overheads.



(a) Example graph. (b) Shards representation.

Figure 1: An example graph partitioned into shards.

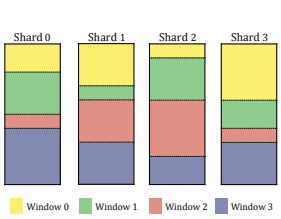
Summary of Results Our experiments with five common graph applications over six real graphs demonstrate that using dynamic shards in GraphChi accelerates the overall processing by up to 2.8× (on average 1.8×). While the accelerated version is still slower than X-Stream in many cases (*cf.* §6.3), this performance gap is reduced by 40% after dynamic partitions are used.

2 The Case for Dynamic Partitions

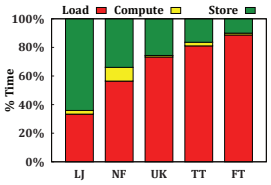
Background A graph $G = (V, E)$ consists of a set of vertices, V, and a set of edges E. The vertices are numbered from 0 to $|V| - 1$. Each edge is a pair of the form $e = (u, v)$, $u, v \in V$. u is the source vertex of e and v is e's destination vertex. e is an incoming edge for v and an outgoing edge for u. The vertex-centric computation model associates a data value with each edge and each vertex; at each vertex, the computation retrieves the values from its incoming edges, invokes an update function on these values to produce the new vertex value, and pushes this value out along its outgoing edges.

The goal of the computation is to "iterate around" vertices to update their values until a global "fixed-point" is reached. There are many programming models developed to support vertex-centric computation, of which the `gather-apply-scatter` (GAS) model is perhaps the most popular one. We will describe the GAS model and how it is adapted to work with dynamic shards in §4. A vertex-centric system iterates around vertices to update their values until a global "fixed-point" is reached.

In GraphChi, the IDs of vertices are split into n disjoint logical intervals, each of which defines a shard. Each shard contains all edge entries whose *target vertices* belong to its defining interval. In other words, the shard only contains incoming edges of the vertices in the interval. As an illustration, given the graph shown in Figure 1a, the distribution of its edges across three shards is shown in Figure 1b where vertices 0–2, 3–5, and 6 are the three intervals that define the shards. If the source of an edge is the same as the previous edge, the edge's *src* field is empty. The goal of such a design is to reduce disk I/O by maximizing sequential disk accesses.

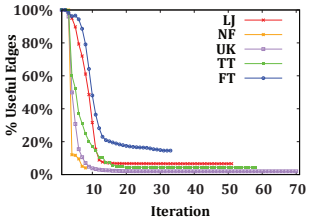


(a) Shards and sliding windows used in GraphChi.

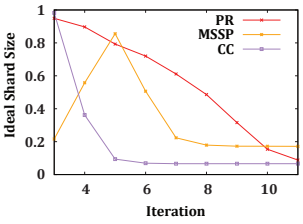


(b) Breakdown of PageRank execution time into load, compute, and store times.

Figure 2: An illustration of sliding windows and the PageRank execution statistics.



(a) Percentages of updated edges across iterations for the PageRank algorithm.



(b) Ideal shard sizes normalized w.r.t. the static shard size for LJ input graph.

Figure 3: Useful data in static shards.

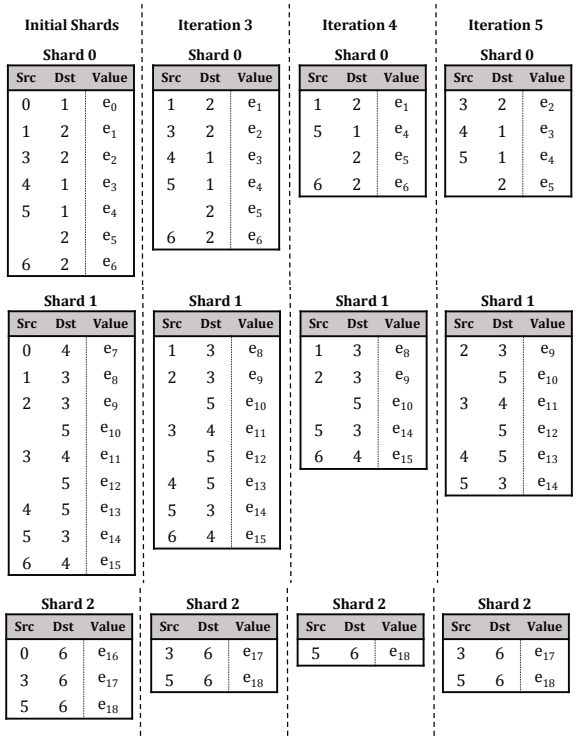


Figure 4: Dynamic shards for the example graph in Figure 1a created for iteration 3, 4 and 5.

Vertex-centric computation requires the presence of all (in and out) edges of a vertex to be in memory when the update is performed on the vertex. Since edges of a vertex may scatter to different shards, GraphChi uses an efficient parallel sliding window (PSW) algorithm to minimize random disk accesses while loading edges. First, edges in a shard s are sorted on their source vertex IDs. This enables an important property: while edges in s can come out of vertices from different intervals, those whose sources are in the same interval i are located contiguously in the shard defined by i.

When vertices v in the interval of s are processed, GraphChi only needs to load s (*i.e.*, memory shard, containing all v's incoming edges and part of v's outgoing edges) and a small block of edges from each other shard (*i.e.*, sliding shard, containing the rest of v's outgoing edges) – this brings into memory a *complete* set of edges for vertices belonging to the interval.

Figure 2a illustrates GraphChi's edge blocks. The four colors are used, respectively, to mark the blocks of edges in each shard whose sources belong to the four intervals defining these shards.

Motivation While the PSW algorithm leverages disk locality, it suffers from redundancy. During computation, a shard contains edges both with and without updated values. Loading the entire shard in every iteration involves wasteful effort of loading and processing edges that are guaranteed to make zero new contribution to the value computation. This effort is significant because (1) the majority of the graph processing cost comes from the loading phase, and (2) at the end of each iteration, there are a large number of edges whose values are unchanged. Figure 2b shows a breakdown of the execution times of PageRank in GraphChi for five real graphs, from the smallest *LiveJournal* (LJ) with 69M edges to *Friendster* (FT) with 2.6B edges. Further details for these input graphs can be found in Table 3.

In these experiments, the I/O bandwidth was fully utilized. Note that the data loading cost increases as the graph becomes larger – for *Friendster*, data loading contributes to over 85% of the total graph processing time. To understand if the impact of data loading is pervasive, we have also experimented with X-Stream [23]. Our results show that the *scatter* phase in X-Stream, which streams all edges in from disk, takes over 70% of the total processing time for PageRank on these five graphs.

To understand how many edges contain necessary data, we calculate the percentages of edges that have updated values across iterations. These percentages are shown in Figure 3a. The percentage of updated edges drops significantly as the computation progresses and becomes very low when the execution comes close to convergence. Significant I/O reductions can be expected if edges not updated in an iteration are completely eliminated from a shard and not loaded in the next iteration.

Figure 3b illustrates, for three applications PageRank (PR), MultipleSourceShortestPath (MSSP), and ConnectedComponents (CC), how the size of an *ideal* shard changes as computation progresses when the LiveJournal graph is processed. In each iteration, an ideal shard only contains edges that have updated values from the previous iteration. Observe that it is difficult to find a one-size-fits-all static partitioning because, for different algorithms, when and where useful data is produced changes dramatically, and thus different shards are needed.

Overview of Techniques The above observations strongly motivate the need for *dynamic shards* whose layouts can be adapted. Conceptually, for each static shard s and each iteration i in which s is processed, there exists a dynamic shard d_i that contains a subset of edges from s whose values are updated in i.

Figure 4 shows the dynamic shards created for Iteration 3, 4 and 5 during the processing of the example graph shown in Figure 1a. After the 2^{nd} iteration, vertex 0 becomes inactive, and hence, its outgoing edges to 4 and 6 are eliminated from the dynamic shards for the 3^{rd} iteration. Similarly, after the 3^{rd} iteration, the vertices 3 and 4 become inactive, and hence, their outgoing edges are eliminated from the shards for Iteration 4. In Iteration 4, the three shards contain only 10 out of a total of 19 edges. Since loading these 10 edges involves much less I/O than loading the static shards, significant performance improvement can be expected. To realize the benefits of dynamic shards by reducing I/O costs, we have developed three techniques:

(1) Processing Dynamic Shards with Delays – Dynamic shards are iteratively processed like static shards; however, due to missing edges in a dynamic shard, we may have to delay the computation of a vertex. We propose a delay based shard processing algorithm that places delayed vertices in an in-memory delay buffer and periodically performs *shadow iterations* that process the delayed requests by bringing in memory all edges for delayed vertices.

(2) Programming Model for Accumulation-Based Computation – Delaying the computation of a vertex if any of its edge is missing can slow the progress of the algorithm. To overcome this challenge we propose an *accumulation-based* programming model that expresses computation in terms of incremental contributions flowing through edges. This maximizes the processing of a vertex by allowing incremental computations to be performed using available edges and thus minimizes the impact of missing edges.

(3) Optimizing Shard Creation – Finally, we develop a practical strategy for balancing the cost of creating dynamic shards with their benefit from reduced I/O by adapting the frequency of shard creation and controlling when a shadow iteration is triggered.

In subsequent sections we discuss each of the above techniques in detail.

3 Processing Dynamic Shards with Delays

Although dynamic shard provides a promising solution to eliminating redundant loading, an immediate question is how to compute vertex values when edges are missing. To illustrate, consider the following graph edges: $u \rightarrow v \rightarrow w$. Suppose in one iteration the value of v is not changed, which means v becomes inactive and the edge $v \rightarrow w$ is not included in the dynamic shard created for the next iteration. However, the edge $u \rightarrow v$ is still included because a new value is computed for u and pushed out through the edge. This value will be reaching v in the next iteration. In the next iteration, the value of v changes as it receives the new contribution from $u \rightarrow v$. The updated value of v then needs to be pushed out through the edge $v \rightarrow w$, which is, however, not present in memory.

To handle missing edges, we allow a vertex to *delay its computation* if it has a missing edge. The delayed computations are batched together and performed in a special periodically-scheduled iteration called *shadow iteration* where all the (in- and out-) edges of the delayed vertices are brought in memory. We begin by discussing dynamic shard creation and then discuss the handling of missing edges.

Creating Dynamic Shards Each computational iteration in GraphChi is divided into three phases: load, compute, and write-back. We build dynamic shards at the end of the compute phase but before write-back starts. In the compute phase, we track the set of edges that receive new values from their source vertices using a *dirty* mark. During write-back, these dirty edges are written into new shards to be used in the next iteration. Evolving graphs can be supported by marking the dynamically added edges to be dirty and writing them into new dynamic shards.

The shard structure has two main properties contributing to the minimization of random disk accesses: (1) *disjoint edge partitioning* across shards and (2) *ordering of edges* based on source vertex IDs inside each shard. Dynamic shards also follow these two properties: since we do not change the logical intervals defined by static partitioning, the edge disjointness and ordering properties are preserved in the newly generated shards. In other words, for each static shard, we generate a dynamic shard, which contains a subset of edges that are stored in the same order as in the static shard. Although our algorithm is inexpensive, creating dynamic shards for every iteration incurs much time overhead and consumes large disk space. We will discuss an optimization in §5 that can effectively reduce the cost of shard creation.

Processing Dynamic Shards Similar to static shards, dynamic shards can be iteratively processed by invoking the user-defined update function on vertices. Although a dynamic shard contains fewer edges than its static counterpart, the logical interval to which the shard belongs is

Algorithm 1 Algorithm for a shadow iteration.

```
1:  S = {S_0, S_1, ..., S_{n-1}}: set of n static shards
2:  DS^i = {DS^i_0, DS^i_1, ..., DS^i_{n-1}}: set of n dynamic shards for
    Iteration i
3:  DS = [DS^0, DS^1, ...]: vector of dynamic shard sets for Iter-
    ation 0, 1, ...
4:  V_i: set of vertex IDs belonging to Interval i
5:  DB: delay buffer containing IDs of the delayed vertices
6:  lastShadow: ID of the last shadow iteration
7:  function SHADOW-PROCESSING(Iteration ite)
8:    for each Interval k from 0 to n do
9:      LOAD-ALL-SHARDS(ite, k)
10:     par-for Vertex v ∈ DB ∩ V_k do
11:       UPDATE(v) //user-defined vertex function
12:     end par-for
13:     produce S'_k by writing updates to the static shard S_k
14:     create a dynamic shard DS^ite_k for the next iteration
15:   end for
16:   remove DS^lastShadow ... DS^(ite-1)
17:   lastShadow ← ite
18:   clear the delay buffer DB
19: end function
20:
21: function LOAD-ALL-SHARDS(Iteration ite, Interval j)
22:   LOAD-MEMORY-SHARD(S_j)
23:   par-for Interval k ∈ [0, n] do
24:     if k ≠ j then
25:       LOAD-SLIDING-SHARD(S_k)
26:     end if
27:   end par-for
28:   for each Iteration k from lastShadow to ite − 1 do =
29:     LOAD-MEMORY-SHARD-AND-OVERWRITE(DS^k_j)
30:     par-for Interval i ∈ [0, n] do
31:       if i ≠ j then
32:         LOAD-SLIDING-SHARD-AND-OVERWRITE(DS^k_i)
33:       end if
34:     end par-for
35:     if k = ite − 1 then
36:       MARK-DIRTY-EDGES( )
37:     end if
38:   end for
39: end function
```

not changed, that is, the numbers of vertices to be updated when a dynamic shard and its corresponding static shard are processed are the same. However, when a dynamic shard is loaded, it contains only *subset* of edges for vertices in its logical interval. To overcome this challenge, we *delay* the computation of a vertex if it has a missing (incoming or outgoing) edge. The delayed vertices are placed in an in-memory delay buffer. We periodically process these delayed requests by bringing in memory all the incoming and outgoing edges for the vertices in the buffer. This is done in a special *shadow iteration* where static shards are also loaded and updated.

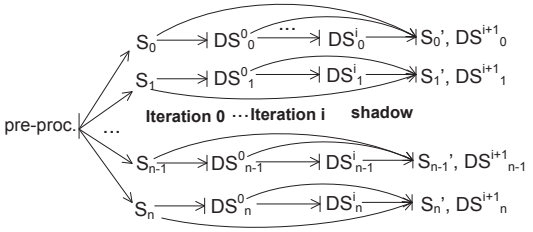


Figure 5: Processing using dynamic shards.

Since a normal iteration has similar semantics as those of iterations in GraphChi, we refer the interested reader to [14] for its details. Here we focus our discussion on shadow iterations. The algorithm of a shadow iteration is shown in Algorithm 1. A key feature of this algorithm is that it loads the static shard (constructed during pre-processing) to which each vertex in the delay buffer belongs to bring into memory all of its incoming and outgoing edges for the vertex computation. This is done by function LOAD-ALL-SHARDS shown in Lines 21–39 (invoked at Line 9).

However, only loading static shards would not solve the problem because they contain out-of-date data for edges that have been updated recently. The most recent data are scattered in the dynamic shards $DS^{lastShadow} \ldots DS^{ite-1}$ where *lastShadow* is the ID of the last shadow iteration and *ite* is the ID of the current iteration. As an example, consider Shard 0 in Figure 4. At the end of iteration 5, the most recent data for the edges $1 \rightarrow 2$, $3 \rightarrow 2$, and $0 \rightarrow 1$ are in DS^4_0, DS^5_0, and S_0, respectively, where DS^i_j represents the dynamic shard for interval j created for iteration i, and S_j denotes the static shard for interval j.

To guarantee that most recent updates are retrieved in a shadow iteration, for each interval j, we sequentially load its static shard S_j (Line 22) and dynamic shards created since the last shadow iteration $DS^{lastShadow} \ldots DS^{ite-1}$ (Line 29), and let the data loaded later *overwrite* the data loaded earlier for the same edges. LOAD-ALL-SHARDS implements GraphChi's PSW algorithm by loading (static and dynamic) memory shards entirely into memory (Lines 22 and 29) and a sliding window of edge blocks from other (static and dynamic) shards (Lines 23–27 and 30–34). If k becomes the ID of the iteration right before the shadow iteration (Lines 35–37), we mark dirty edges to create new dynamic shards for the next iteration (Line 14).

After the loop at Line 8 terminates, we remove all the intermediate dynamic shards (Line 16) and set *lastShadow* to *ite* (Line 17). These shards are no longer needed, because the static shards are already updated with the most recent values in this iteration (Line 13). One can think of static shards as "checkpoints" of the computation and dynamic shards as intermediate "increments" to the most recent checkpoint. Finally, the delay buffer is cleared.

Figure 5 illustrates the input and output of each computational iteration. Static shards $S_0 \ldots S_n$ are statically

constructed. Each regular iteration i produces a set of dynamic shards $DS_0^i \ldots DS_n^i$, which are fed to the next iteration. A shadow iteration loads all static shards and intermediate dynamic shards, and produces (1) updated static shards $S'_0 \ldots S'_n$ and (2) new dynamic shards $DS_0^{i+1} \ldots DS_n^{i+1}$ to be used for the next iteration.

It may appear that the delay buffer can contain many vertices and consume much memory. However, since the amount of memory needed to represent an incoming edge is higher than that to record a vertex, processing dynamic shards with the delay buffer is actually more memory-efficient compared to processing static shards where all edges are available.

Delaying a vertex computation when *any* of its edge is missing can cause too many vertices to be delayed and negatively impact the the computation progress. For example, when running PageRank on *UKDomain*, *Twitter*, and *Friendster* graphs, immediately after the dynamic shards are created, 64%, 70%, and 73% of active vertices are delayed due to at least one missing incoming or outgoing edge. Frequently running shadow iterations may get data updated quickly at the cost of extra overhead, while doing so infrequently would reduce overhead but slow down the convergence. Hence, along with dynamically capturing the set of edges which reflect change in values, it is important to modify the computation model so that it maximizes computation performed using available values.

§4 presents an optimization for our delay-based computation to limit the number of delayed computations. The optimization allows a common class of graph algorithms to perform vertex computation if a vertex only has missing incoming edges. While the computation for vertices with missing outgoing edges still need to be delayed, the number of such vertices is much smaller, leading to significantly reduced delay overhead.

4 Accumulation-based Computation

This section presents an *accumulation-based* programming/execution model that expresses computation in terms of *incremental contributions* flowing through edges. Our insight is that if a vertex is missing an incoming edge, then the edge is guaranteed to provide zero *new contribution* to the vertex value. If we can design a new model that performs updates based on *contribution increments* instead of actual contributions, the missing incoming edge can be automatically treated as *zero* increment and the vertex computation can be performed without delay.

We discuss our approach based on the popular Gather-Apply-Scatter (GAS) programming model [14, 16, 8]. In the GAS model, vertex computation is divided in three distinct phases: the `gather` phase reads incoming edges and produces an aggregated value using a user-defined aggregation function; this value is fed to the `apply` phase to compute a new value for a vertex; in the `scatter` phase, the value is propagated along the outgoing edges.

4.1 Programming Model

Our accumulation-based model works for a common class of graph algorithms whose GAS computation is *distributive* over aggregation. The user needs to program the GAS functions in a slightly different way to propagate *changes in values* instead of actual values. In other words, the semantics of vertex data remains the same while data on each edge now encodes the delta between the old and the new value of its source vertex. This semantic modification relaxes the requirement that all incoming edges of a vertex have to be present to perform vertex computation.

The new computation semantics requires minor changes to the GAS programming model. (1) `Extract` the gathered value using the old vertex value. This step is essentially an inverse of the `apply` phase that uses its output (*i.e.*, vertex value) to compute its input (*i.e.*, aggregated value). (2) `Gather` edge data (*i.e.*, from present incoming edges) and aggregate it together with the output of `extract`. Since this output represents the contributions of the previously encountered incoming edges, this step incrementally adds new contributions from the present incoming edges to the old contributions. (3) `Apply` the new vertex value using the output of `gather`. (4) `Scatter` the difference between the old and the new vertex values along the outgoing edges.

To turn a GAS program into a new program, one only needs to add an `extract` phase in the beginning that uses a vertex value v to compute *backward* the value g gathered from the incoming edges of the vertex at the time v was computed. g is then aggregated with a value gathered from the present incoming edges to compute a new value for the vertex. To illustrate, consider the PageRank algorithm that has the following GAS functions:

$$\begin{aligned}
&\text{[GATHER]} && sum \leftarrow \Sigma_{e \in in(v)} e.data \\
&\text{[APPLY]} && v.pr \leftarrow (0.15 + 0.85 * sum)/ \\
& && \quad v.numOutEdges \\
&\text{[SCATTER]} && \forall e \in out(v) : e.data \leftarrow v.pr
\end{aligned}$$

Adding the `extract` phase produces:

$$\begin{aligned}
&\text{[EXTRACT]} && oldsum \leftarrow (v.pr * v.numOutEdges \\
& && \quad - 0.15)/0.85 \\
&\text{[GATHER]} && newsum \leftarrow oldsum + \Sigma_{e \in in(v)} e.data \\
&\text{[APPLY]} && newpr \leftarrow (0.15 + 0.85 \times newsum)/ \\
& && \quad v.numOutEdges; \\
& && oldpr \leftarrow v.pr;\ \ v.pr \leftarrow newpr \\
&\text{[SCATTER]} && \forall e \in out(v) : e.data \leftarrow newpr - oldpr
\end{aligned}$$

In this example, `extract` reverses the PageRank computation to obtain the old aggregated value *oldsum*, on top of which the new contributions of the present incoming edges are added by `gather`. `Apply` keeps its original semantics and computes a new PageRank value. Before this new value is saved on the vertex, the delta between the old and new is computed and propagated along the outgoing edges in `scatter`.

An alternative way to implement the accumulation-based computation is to save the value gathered from incoming edges on each vertex (*e.g.*, *oldsum*) together with the vertex value so that we do not even need the `extract` phase. However, this approach doubles the size of vertex data which also negatively impacts the time cost due to the extremely large numbers of vertices in real-world graphs. In fact, the `extract` phase does not create extra computation in most cases: after simplification and redundancy elimination, the PageRank formulas using the traditional GAS model and the accumulation-based model require the same amount of computation:

$$pr = \begin{cases} 0.15 + 0.85 \times sum & \ldots \text{traditional} \\ v.pr + 0.85 \times sum & \ldots \text{accumulation-based} \end{cases}$$

Impact on the Delay Buffer Since the contribution of each incoming edge can be incrementally added onto the vertex value, this model does not need the presence of all incoming edges to compute vertex values. Hence, it significantly decreases the number of vertices whose computation needs to be delayed, reducing the need to frequently run shadow iterations.

If a vertex has a missing outgoing edge, delay is needed. To illustrate, consider again the $u \to v \to w$ example in the beginning of §3. Since the edge $v \to w$ is missing, although v gets an updated value, the value cannot be pushed out. We have to delay the computation until a shadow iteration in which $v \to w$ is brought into memory. More precisely, v's `gather` and `apply` can still be executed right away; only its `scatter` operation needs to be delayed, because the target of the `scatter` is unknown due to the missing outgoing edge.

Hence, for each vertex, we execute `gather` and `apply` instantly to obtain the result value r. If the vertex has a missing outgoing edge, the vertex is pushed into the delay buffer together with the value r. Each entry in the buffer now becomes a vertex-value pair. In the next shadow iteration, when this missing edge is brought into memory, r will be pushed through the edge and be propagated.

Since a vertex with missing outgoing edges can be encountered multiple times before a shadow iteration is scheduled, the delay buffer may contain multiple entries for the same vertex, each with a different delta value. Naïvely propagating the most recent increment is incorrect due to the accumulative nature of the model; the consideration of *all* the entries for the vertex is thus required. Hence, we require the developer to provide an additional *aggregation function* that takes as input an ordered list of all delta values for a vertex recorded in the delay buffer and generates the final value that can be propagated to its outgoing edges (details are given in §4.2).

Although our programming model exposes the `extract` phase to the user, not all algorithms need this phase. For example, algorithms such as ShortestPath and ConnectedComponents can be easily coded in a traditional way, that is, edge data still represent actual values (*i.e.*, paths or component IDs) instead of value changes. This is because in those algorithms, vertex values are in *discrete domains* and `gather` is done by monotonically selecting a value from one incoming edge instead of accumulating values from all incoming edge values. For instance, ShortestPath and ConnectedComponents use selection functions (*min*/*max*) to aggregate contributions of incoming edges.

To make the differences between algorithm implementations transparent to the users, we allow users to develop normal GAS functions without thinking about what data to push along edges. The only additional function the user needs to add is `extract`. Depending on whether `extract` is empty, our system *automatically* determines the meaning of edge data and how it is pushed out.

4.2 Model Applicability and Correctness

It is important to understand precisely what algorithms can and cannot be implemented under the accumulation-based model. There are three important questions to ask about applicability: (1) what is the impact of *incremental computation* on graph algorithms, (2) what is the impact of *delay* on those algorithms, and (3) is the computation still correct when vertex updates are delayed?

Impact of Incremental Computation An algorithm can be correctly implemented under our accumulation-based model if the composition of its `apply` and `gather` is distributive on some aggregation function. More formally, if vertex v has n incoming edges $e_1, e_2, \ldots e_n$, v's computation can be expressed under our accumulation-based model iff there exists an aggregation function[1] f *s.t.*

$$\texttt{apply}(\texttt{gather}(e_1, \ldots, e_n)) = f(\texttt{apply}(\texttt{gather}(e_1)), \ldots, \texttt{apply}(\texttt{gather}(e_n)))$$

For most graph algorithms, we can easily find a function f on which their computation is distributive. Table 1 shows a list of 24 graph algorithms studied in recent graph papers and our accumulation-based model works for all but two. For example, one of these two algorithms is GraphColoring, where the color of a vertex is determined

[1] The commutative and associative properties from `gather` get naturally lifted to the aggregation function f.

	V/E	Iteration 0	1	2	3	4 (Shadow)
	u	$[0, I_u]$	$[I_u, I_u]$	$[I_u, a]$	$[a, b]$	$[b, x]$
No Delay	$u \to v$	$[0, I_u]$	$[I_u, 0]$	$[0, a - I_u]$	$[a - I_u, b - a]$	$[b - a, x - b]$
	v	$[0, I_v]$	$[I_v$, AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+a)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+a)]	[AP(EX(I_v)+a), AP(EX(I_v)+b)]
Delay	$u \to v$	$[0, I_u]$	$[I_u, 0]$	Missing	Missing	$[b - I_u, x - b]$
	v	$[0, I_v]$	[I_v, AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+I_u)]	[AP(EX(I_v)+I_u), AP(EX(I_v)+b)]

Table 2: A comparison between PageRank executions with and without delays under the accumulation-based model; for each vertex and edge, we use a pair [a, b] to report its pre- (a) and post-iteration (b) value. Each vertex u (v) has a value 0 before it receives an initial value I_u (I_v) in Iteration 0; `EX` and `AP` represent function `Extract` and `Apply`, respectively.

Algorithms	**Aggr. Func.** f
Reachability, MaxIndependentSet	or
TriangleCounting, SpMV, PageRank, HeatSimulation, WaveSimulation, NumPaths	sum
WidestPath, Clique	max
ShortestPath, MinmialSpanningTree, BFS, ApproximateDiameter, ConnectedComponents	min
BeliefPropagation	product
BetweennessCentrality, Conductance, NamedEntityRecognition, LDA, ExpectationMaximization, AlternatingLeastSquares	user-defined aggregation function
GraphColoring, CommunityDetection	N/A

Table 1: A list of algorithms used as subjects in the following papers and their aggregation functions if implemented under our model: GraphChi [14], GraphLab [16], ASPIRE [25], X-Stream [23], GridGraph [31], GraphQ [27], GraphX [9], PowerGraph [8], Galois [19], Ligra [24], Cyclops [5], and Chaos [22].

by the colors of all its neighbors (coming through its incoming edges). In this case, it is impossible to compute the final color by applying `gather` and `apply` on different neighbors' colors separately and aggregating these results. For the same reason CommunityDetection cannot be correctly expressed as an incremental computation.

Once function f is found, it can be used to aggregate values from multiple entries of the same vertex in the delay buffer, as described earlier in §4.1. We provide a set of built-in f from which the user can choose, including *and*, *or*, *sum*, *product*, *min*, *max*, *first*, and *last*. For instance, PageRank uses *sum* that produces the final delta by summing up all deltas in the buffer, while ShortestPath only needs to compute the minimum of these deltas using *min*. The user can also implement her own for more complicated algorithms that perform numerical computations.

For graph algorithms with non-distributive `gather` and `apply`, using dynamic partitions has to delay computation for a great number of vertices, making overhead outweigh benefit. In fact, we have implemented GraphColoring in our system and only saw slowdowns in the experiments. Hence, our optimization provides benefit only for distributive graph algorithms.

Impact of Delay To understand the impact of delay, we draw a connection between our computation model with the staleness-based (*i.e.*, relaxed consistency) computation model [25, 6]. The staleness-based model allows computation to be performed on stale values but guarantees correctness by ensuring that all updates are visible at some point during processing (by either using refresh or imposing a staleness upper-bound). This is conceptually similar to our computation model with delays: for vertices with missing outgoing edges, their out-neighbors would operate on stale values until the next shadow iteration.

Since a shadow iteration "refreshes" all stale values, the frequency of performing these shadow iterations bounds the maximum staleness of edge values. Hence, any algorithm that can correctly run under the relaxed consistency model can also safely run under our model. Moreover, the frequency of shadow iterations has no impact on the correctness of such algorithms, as long as they do occur and flush the delayed updates. In fact, all the algorithms in Table 1 would function correctly under our delay-based model. However, their performance can be degraded if they cannot employ incremental computation.

Delay Correctness Argument While our delay-based model shares similarity with the staleness-based model, the correctness of a specific algorithm depends on the aggregation function used for the algorithm. Here we provide a correctness argument for the aggregation functions we developed for the five algorithms used in our evaluation: PageRank, BeliefPropagation, HeatSimulation, ConnectedComponents, and MultipleSourceShortestPath; similar arguments can be used for other algorithms in Table 1.

We first consider our implementation of PageRank that propagates changes in page rank values along edges. Since BeliefPropagation and HeatSimulation perform similar computations, their correctness can be reasoned in the same manner. For a given edge $u \to v$, Table 2 shows, under the accumulation-based computation, how the val-

ues carried by vertices and edges change across iterations with and without delays.

We assume that each vertex u (v) has a value 0 before it is assigned an initial value I_u (I_v) in Iteration 0 and vertex v has only one incoming edge $u \rightarrow v$. At the end of Iteration 0, both vertices have their initial values because the edge does not carry any value in the beginning. We further assume that in Iteration 1, the value of vertex u does not change. That is, at the end of the iteration, u's value is still I_u and, hence, the edge will not be loaded in Iteration 2 and 3 under the delay-based model.

We compare two scenarios in which delay is and is not enabled and demonstrate that the same value is computed for v in both scenarios. Without delay, the edge value in each iteration always reflects the change in u's values. v's value is determined by the four functions described earlier. For example, since the value carried by the edge at the end of Iteration 0 is I_u, v's value in Iteration 1 is updated to `apply(gather(extract(`I_v`), `I_u`))`. As `gather` is `sum` in PageRank, this value reduces to `AP(EX(`I_v`) + `I_u`)`. In Iteration 2, the value from the edge is 0 and thus v's value becomes `AP(EX(AP(EX(`I_v`) + `I_u`)) + 0)`. Because `EX` is an inverse function of `AP`, this value is thus still `AP(EX(`I_v`) + `I_u`)`. Using the same calculation, we can easily see that in Iteration 4 v's value is updated to `AP(EX(`I_v`) + `b`)`.

With delay, the edge will be missing in Iteration 2 and 3, and hence, we add two entries (u, $a - I_u$) and (u, $b - a$) into the delay buffer. During the shadow iteration, the edge is loaded back into memory. The aggregation function `sum` is then applied on these two entries, resulting in value $b - I_u$. This value is pushed along $u \rightarrow v$, leading to the computation of the following value for v:

$$\begin{aligned} & \texttt{AP(EX(AP(EX(}I_v\texttt{) + }I_u\texttt{)) + (}b - I_u\texttt{))} \\ \Rightarrow\ & \texttt{AP(EX(}I_v\texttt{) + }I_u + b - I_u\texttt{)} \\ \Rightarrow\ & \texttt{AP(EX(}I_v\texttt{) + }b\texttt{)} \end{aligned}$$

which is the same as the value computed without delay.

This informal correctness argument can be used as the base case for a formal proof by induction on iterations. This proof is omitted from the paper due to space limitations. Although we have one missing edge in this example, the argument can be easily extended to handle multiple missing edges since the `gather` function is associative.

For ShortestPaths and ConnectedComponents, they do not have an `extract` function and their contributions are gathered by the selection function *min*. Since a dynamic shard can never have edges that are not part of its corresponding static shard, vertex values (*e.g.*, representing path and component IDs) in the presence of missing edges are always greater than or equal to their actual values. It is thus easy to see that the aggregation function *min* ensures that during the shadow iteration the value a of each vertex will be appropriately overridden by the minimum value b of the delayed updates for the vertex if $b \leq a$.

4.3 Generalization to Edge-Centricity

Note that the dynamic partitioning techniques presented in this work can be easily applied to edge-centric systems. For example, X-Stream [23] uses an unordered edge list and a scatter-gather computational model, which first streams in the edges to generate updates, and then streams in the generated updates to compute vertex values. To enable dynamic partitioning, dynamic edge lists can be constructed based on the set of changed vertices from the previous iterations. This can be done during the scatter phase by writing to disk the required edges whose vertices are marked dirty.

Hence, later iterations will stream in smaller edge lists that mainly contain the necessary edges. Similarly to processing dynamic shards, computations in the presence of missing edges can be delayed during the gather phase when the upcoming scatter phase cannot stream in the required edges. These delayed computations can be periodically flushed by processing them during shadow iterations in which the original edge list is made available.

GridGraph [31] is a recent graph system that uses a similar graph representation as used in GraphChi. Hence, our shard-based techniques can be applied directly to partitions in GridGraph. As GridGraph uses very large static partitions (that can accommodate tens of millions of edges), larger performance benefit may be seen if our optimization is added. Dynamic partitions can be generated when edges are streamed in; computation that needs to be delayed due to missing edges can be detected when vertices are streamed in.

5 Optimizing Shard Creation

To maximize net gains, it is important to find a sweet spot between the cost of creating a dynamic shard and the I/O reduction it provides. This section discusses an optimization and analyzes its performance benefit.

5.1 Optimization

Creating a dynamic shard at each iteration is an overkill because many newly created dynamic shards provide only small additional reduction in I/O that does not justify the cost of creating them. Therefore, we create a new dynamic shard after several iterations, allowing the creation overhead to be easily offset by the I/O savings.

Furthermore, to maximize edge reuse and reduce delay frequencies, it is useful to include into dynamic shards edges that may be used in *multiple* subsequent iterations. We found that using shadow iterations to create dynamic shards strikes a balance between I/O reduction and overhead of delaying computations – new shards are created only during shadow iterations; we treat edges that were updated after the previous shadow iteration as dirty and

include them all in the new dynamic shards. The intuition here is that by considering an "iteration window" rather than one single iteration, we can accurately identify edges whose data have truly stabilized, thereby simultaneously reducing I/O and delays.

The first shadow iteration is triggered when the percentage of updated edges p in an iteration drops below a threshold value. The frequency of subsequent shadow iterations depends upon the size of the delay buffer d — when the buffer size exceeds a threshold, a shadow iteration is triggered. Hence, the frequency of shard creation is adaptively determined, in response to the progress towards convergence. We used $p = 30\%$ and $d = 100$KB in our experiments and found them to be effective.

5.2 I/O Analysis

We next provide a rigorous analysis of the I/O costs. We show that the overhead of shard loading in shadow iterations can be easily offset from the I/O savings in regular non-shadow iterations. We analyze the I/O cost in terms of the number of data blocks transferred between disk and memory. Let b be the size of a block in terms of the number of edges and E be the edge set of the input graph. Let AE_i (*i.e.*, active edge set) represent the set of edges in the dynamic shards created for iteration i. Here we analyze the cost of regular iterations and shadow iterations separately for iteration i.

During regular iterations, processing is done using the static shards in the first iteration and most recently created dynamic shards during later iterations. Each edge can be read at most twice (*i.e.*, when its source and target vertices are processed) and written once (*i.e.*, when the value of its source vertex is pushed along the edge). Thus,

$$C_i \leq \begin{cases} \frac{3|E|}{b} & \text{with static shards} \\ \frac{3|AE_i|}{b} & \text{with dynamic shards} \end{cases} \quad (1)$$

In a shadow iteration, the static shards and all intermediate dynamic shards are read, the updated edges are written back to static shards, and a new set of dynamic shards are created for the next iteration. Since we only append edges onto existing dynamic shards in regular iterations, there is only one set of dynamic shards between any consecutive shadow iterations. Hence, the I/O cost is:

$$C_i \leq \frac{3|E|}{b} + \frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b} \quad (2)$$

where AE_{LS} is the set of edges in the dynamic shards created by the last shadow iteration. Clearly, C_i is larger than the cost of static shard based processing (*i.e.*, $\frac{3|E|}{b}$).

Eq. 1 and Eq. 2 provide a useful insight on how the overhead of a shadow iteration can be amortized across regular iterations. Based on Eq. 2, the extra I/O cost of a shadow iteration over a regular static-shard-based iteration is $\frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b}$. Based on Eq. 1, the I/O saving achieved by using dynamic shards in a regular iteration is $(\frac{3|E|}{b} - \frac{3|AE_i|}{b})$.

We assume that d shadow iterations have been performed before the current iteration i and hence, the frequency of shadow iterations is $\frac{i}{d}$ (for simplicity, we assume i is multiple of d). This means, shadow iteration occurs once every $\frac{i}{d} - 1$ regular iterations.

In order for the overhead of a shadow iteration to be wiped off by the savings in regular iterations, we need:

$$(\frac{i}{d} - 1) \times (\frac{3|E|}{b} - \frac{3|AE_i|}{b}) \geq \frac{2|AE_{LS}|}{b} + \frac{|AE_i|}{b}$$

After simplification, we need to show:

$$(\frac{i}{d} - 1) \times 3|E| - (\frac{3i}{d} - 2) \times |AE_i| - 2|AE_{LS}| \geq 0 \quad (3)$$

Since AE_{LS} is the set of edges in the dynamic shards before Iteration i, we have $|AE_{LS}| \leq |AE_i|$ as we only append edges after that shadow iteration. We thus need to show:

$$(\frac{i}{d} - 1) \times 3|E| - (\frac{3i}{d} - 2) \times |AE_i| - 2|AE_i| \geq 0$$
$$\Longrightarrow \frac{|E|}{|AE_i|} \geq \frac{\frac{i}{d}}{\frac{i}{d} - 1}$$

The above inequality typically holds for any frequency of shadow iterations $\frac{i}{d} > 1$. For example, if the frequency of shadow iterations $\frac{i}{d}$ is 3, $\frac{|E|}{|AE_i|} \geq 1.5$ means that as long as the total size of static shards is 1.5 times larger than the total size of (any set of) dynamic shards, I/O efficiency can be achieved by our optimization. As shown in Figure 3a, after about 10 iterations, the percentage of updated edges in each iteration goes below 15%. Although unnecessary edges are not removed in each iteration, the ratio between $|E|$ and $|AE_i|$ is often much larger than 1.5, which explains the I/O reduction from a theoretical viewpoint.

6 Evaluation

Our evaluation uses five applications including PageRank (PR) [20], MultipleSourceShortestPath (MSSP), Belief-Propagation (BP) [11], ConnectedComponents (CC) [30], and HeatSimulation (HS). They belong to different domains such as social network analysis, machine learning, and scientific simulation. They were implemented using our accumulation-based GAS programming model. Six real-world graphs, shown in Table 3, were chosen as inputs for our experiments.

All experiments were conducted on an 8-core commodity Dell machine with 8GB main memory, running Ubuntu 14.04 kernel 3.16, a representative of low-end PCs regular users have access to. Standard Dell 500GB 7.2K RPM HDD and Dell 400GB SSD were used as secondary

Inputs	Type	#Vertices	#Edges	PMSize	#SS
LiveJournal (LJ) [2]	Social Network	4.8M	69M	1.3GB	3
Netflix (NF) [3]	Recomm. System	0.5M	99M	1.6GB	20
UKDoman (UK) [4]	Web Graph	39.5M	1.0B	16.9GB	20
Twitter (TT) [13]	Social Network	41.7M	1.5B	36.3GB	40
Friendster (FT) [7]	Social Network	68.3M	2.6B	71.6GB	80
YahooWeb (YW) [28]	Web Graph	1.4B	6.6B	151.3GB	120

Table 3: Input graphs used; **PMSize** and **SS** report the peak in-memory size of each graph structure (without edge values) and the number of static shards created in GraphChi, respectively. The in-memory size of a graph is measured as the maximum memory consumption of a graph across the five applications; LJ and NF are relatively small graphs while UK, TT, FT, YW are **billion-edge graphs larger than the 8GB memory size**; YW is the **largest real-world graph** publicly available; all graphs have highly skewed power-law degree distributions.

G	Version	PR	BP	HS	MSSP	CC
LJ	BL	630	639	905	520	291
	ADS	483	426	869	535	296
	ODS	258	383	321	551	263
NF	BL	189	876	238	1,799	190
	ADS	174	597	196	1,563	177
	ODS	158	568	164	1,436	178
UK	BL	31,616	19,486	21,620	74,566	14,346
	ADS	23,332	15,593	35,200	76,707	14,742
	ODS	14,874	14,227	12,388	67,637	12,814
TT	BL	83,676	47,004	75,539	109,010	22,650
	ADS	61,994	38,148	67,522	97,132	21,522
	ODS	47,626	28,434	30,601	84,058	21,589
FT	BL	130,928	100,690	159,008	146,518	50,762
	ADS	85,788	84,502	176,767	143,798	50,831
	ODS	87,112	51,905	63,120	127,168	42,956

Table 4: A comparison on execution time (seconds) among **Baseline** (BL), **ADS**, and **ODS**.

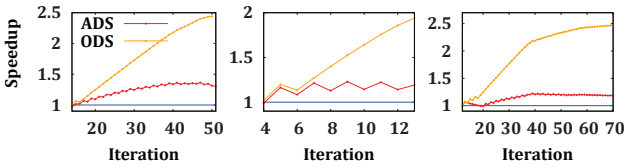
storage, both of which were connected via SATA 3.0Gb/s interface. File system caches were flushed before running experiments to make different executions comparable.

Two relatively small graphs LJ and NF were chosen to understand the scalability trend of our technique. The other four graphs UK, TT, FT, and YW are larger than memory by 2.4×, 5.2×, 10.2×, and 21.6× respectively.

6.1 Overall Performance

We compared our modified GraphChi extensively with the **Baseline (BL)** GraphChi that processes static shards in parallel. To provide a better understanding of the impact of the shard creation optimization stated in §5, we made two modifications, one that creates dynamic shards aggressively (**ADS**) and a second that uses the optimization in §5 (**ODS**). We first report the performance of our algorithms over the first five graphs on HDD in Table 4.

We ran each program until it converged to evaluate the full impact of our I/O optimization. We observed that for each program the numbers of iterations taken by **Baseline**



(a) PR on LJ. (b) BP on FT. (c) HS on TT.

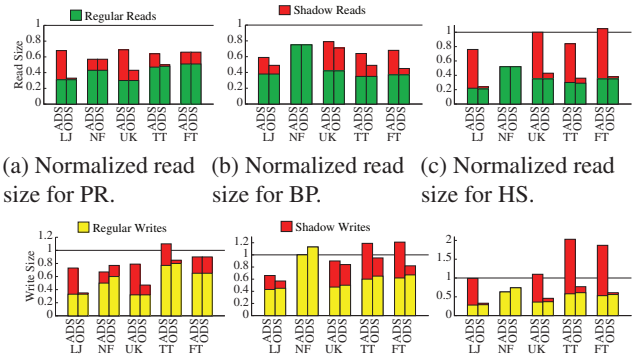
Figure 6: Speedups achieved per iteration.

and **ODS** are almost the same. That is, despite the delays needed due to missing edges, the accumulation-based computation and shard creation optimizations minimize the vertices that need to be delayed, yielding the same convergence speed in **ODS**. **ADS** can increase the number of iterations in a few cases due to the delayed convergence. Due to space limitations, the iteration numbers are omitted from the paper. On average, **ADS** and **ODS** achieve an up to 1.2× and 1.8× speedup over *Baseline*.

PR, BP, and HS are computation-intensive programs and they operate on large working sets. For these three programs, on average **ADS** speeds up graph processing by 1.53×, 1.50× and 1.22×, respectively. **ODS** performs much better providing speedups of 2.44×, 1.94×, and 2.82× respectively. The optimized version **ODS** performs better than the aggressive version **ADS** because **ODS** is likely to eliminate edges after the computation of their source vertices becomes stable, and thus edges that will be useful in a few iterations are likely to be preserved in dynamic shards. **ODS** consistently outperforms the baseline. While **ADS** outperforms the baseline in most cases, eliminating edges aggressively delays the algorithm convergence for HS on UK (*i.e.*, by 20% more iterations).

MSSP and CC require less computation and they operate on smaller and constantly changing working sets. Small benefits were seen from both **ADS** (1.15× speedup) and **ODS** (1.30× speedup), because eliminating edges achieves I/O efficiency at the cost of locality.

Figure 6 reports a breakdown of speedups on iterations for PR, BP, and HS. Two major observations can



(a) Normalized read size for PR. (b) Normalized read size for BP. (c) Normalized read size for HS.

(d) Normalized write size for PR. (e) Normalized write size for BP. (f) Normalized write size for HS.

Figure 7: Read and write size for different benchmarks normalized w.r.t. the baseline.

be made here. First, the performance improvement increases as the computation progresses, which confirms our intuition that the amount of useful data decreases as the computation comes close to the convergence. Second, the improvements from **ADS** exhibit a saw-tooth curve, showing the need of the optimizations in **ODS**: frequent drops in speedups are due to frequent shard creation and shadow iterations. These time reductions are entirely due to reduced I/O because the numbers of iterations taken by **ODS** and **Baseline** are almost always the same.

	PR	BP	HS
BL	153h:33m	80h:19m	147h:48m
ODS	92h:26m	54h:29m	92h:7m
Speedup	**1.66**×	**1.47**×	**1.60**×

Table 5: PR, BP and HS on YW.

Since the YW graph is much larger and takes much longer to run, we evaluate **ODS** for PR, BP and HS whose performance is reported in Table 5. **ODS** achieves a 1.47 – 1.60× speedup over **Baseline** for PR, BP and HS.

Performance on SSD To understand whether the proposed optimization is still effective when high-bandwidth **SSD** is used, we ran experiments for PR and BP on a machine with the same configuration except that SSD is employed to store shards. We found that the performance benefits are consistent when SSD is employed: on average, **ADS** accelerates PR, BP and HS by 1.25×, 1.18× and 1.14× respectively, whereas **ODS** speeds them up by 1.67×, 1.52× and 1.91× respectively.

Our techniques are independent of the storage type and the performance benefits are mainly achieved by reducing shard loading time. This roughly explains why a lower benefit is seen on SSD than on HDD – for example, compared to HDD, the loading time for FT on SSD decreases by 8%, 11% and 7% for PR, BP and HS, respectively.

6.2 I/O Analysis

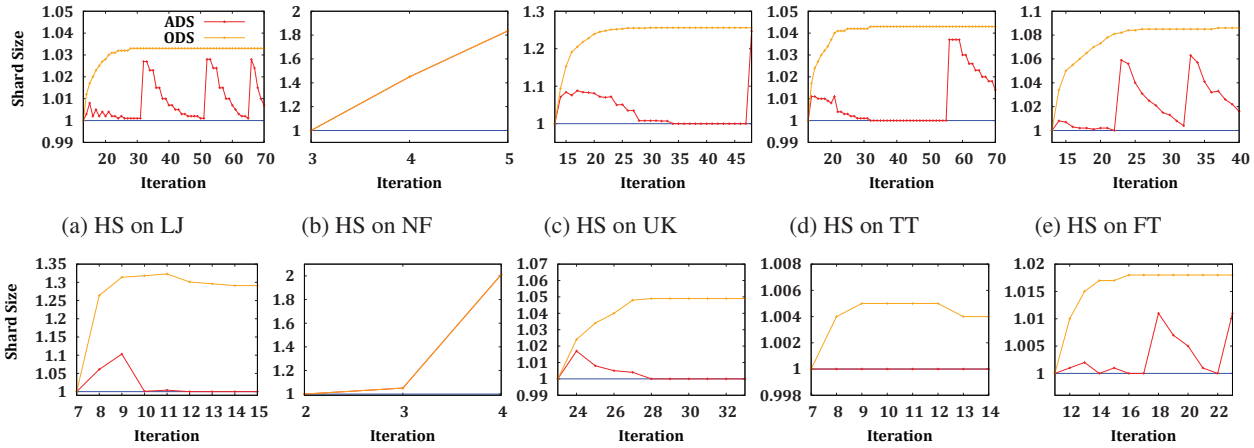
Data Read/Written Figure 7 shows the amount of data read and written during the graph processing in the modified GraphChi, normalized *w.r.t. Baseline*. Reads and writes that occur during shadow iterations are termed *shadow reads* and *shadow writes*. No shadow iteration has occurred when some applications were executed on the Netflix graph (*e.g.*, in Figures 7 (b), (c), (e), and (f)), because processing converges quickly and dynamic shards created once are able to capture the active set of edges until the end of execution.

Due to space limitations, we only show results for PR, BP and HS; similar observations can be made for the other applications. Clearly, **ODS** reads/writes much less data than both **Baseline** and **ADS**. Although shadow iterations incur additional I/O, this overhead can be successfully offset from the savings in regular iterations. **ADS** needs to read and write more data than **Baseline** in some cases (*e.g.*, Friendster in Figure 7c, Twitter in Figure 7d and Figure 7e). This shows that creating dynamic shards too frequently can negatively impact performance.

Size of Dynamic Shards To understand how well **ADS** and **ODS** create dynamic shards, we compare the sizes of intermediate dynamic shards created using these two strategies. Figure 8 shows the change of the sizes of dynamic shards as the computation progresses, normalized *w.r.t.* the size of an ideal shard. The ideal shard for a given iteration includes only the edges which were updated in the previous iteration, and hence, it contains the minimum set of edges necessary for the next iteration. Note that for both **ADS** and **ODS**, their shard sizes are close to the ideal sizes. In most cases, the differences are within 10%.

It is also expected that shards created by **ODS** are often larger than those created by **ADS**. Note that patterns exist in shard size changes for **ADS** such as HS on LJ (Figure 8a) and FT (Figure 8e). This is because the processing of delayed operations (in shadow iterations) over high-degree vertices causes many edges to become active and be included in new dynamic shards.

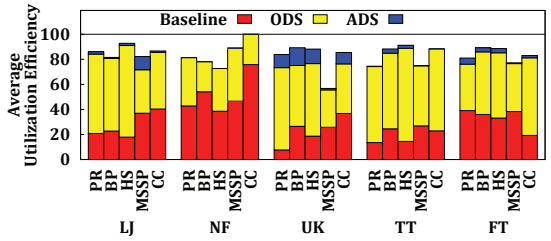
Edge Utilization Figure 9a reports the average *edge utilization rates* (EUR) for **ADS** and **ODS**, and compares them with that of **Baseline**. The average edge utilization rate is defined as the percentage of updated edges in a dynamic shard, averaged across iterations. Using dynamic shards highly improves the edge utilization: the EURs for **ADS** and **ODS** are between 55% and 92%. For CC on NF, the utilization rate is 100% even for **ODS**, because computation converges quickly and dynamic shards are created only once. Clearly, **ADS** has higher EURs than **ODS** because of its aggressive shard creation strategy. Using static shards throughout the execution leads to a very low EUR for **Baseline**.



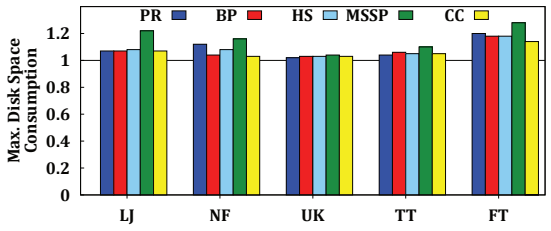
(a) HS on LJ (b) HS on NF (c) HS on UK (d) HS on TT (e) HS on FT

(f) MSSP on LJ (g) MSSP on NF (h) MSSP on UK (i) MSSP on TT (j) MSSP on FT

Figure 8: The dynamic shard sizes normalized *w.r.t.* the ideal shard sizes as the algorithm progresses.



(a) Edge utilization rates.



(b) Max disk space used.

Figure 9: Edge and disk utilization statistics.

Disk Space Consumption Figure 9b reports the maximum disk space needed to process dynamic shards normalized *w.r.t.* that needed by **Baseline**. Since we create and use dynamic shards only after vertex computations start stabilizing, the actual disk space it requires is very close to (but higher than) that required by **Baseline**. This can be seen in Figure 9b where the disk consumption increases by 2-28%. Note that the maximum disk space needed is similar for **ADS** and **ODS**, because dynamic shards created for the first time take most space; subsequent shards are either smaller (for **ADS**), or additionally include a small set of active edges (for **ODS**), which is insignificant to affect the ratio.

Delay Buffer Size With the help of the accumulation-based computation, the delay buffer often stays small throughout the execution. Its size is typically less than few 100KBs. The peak consumption was seen when ConnectedComponent was run on the Friendster graph, and the buffer size was 1.5MB.

6.3 Comparisons with X-Stream

Figure 10 compares the speedups and the per-iteration savings achieved by **ODS** and **X-Stream** over **Baseline** when running PR on large graphs. The saving per iteration was obtained by (1) calculating, for each iteration in which dynamic shards are created, $\frac{Baseline-ODS}{Baseline}$, and (2) taking an average across savings in all such iterations. While the per-iteration savings achieved by dynamic shards are higher than those by X-Stream, **ODS** is overall slower than X-Stream (*i.e.*, **ODS** outperforms X-Stream on UK but underperforms it on other graphs).

This is largely expected due to the fundamentally different designs of the vertex- and edge-centric computation models. Our optimization is implemented in GraphChi, which is designed to scan the whole graph multiple times during each iteration, while X-Stream streams edges in and thus only needs one single scan. Hence, although our optimization reduces much of GraphChi's loading time, this reduction is not big enough to offset the time spent on extra graph scans. Furthermore, in order to avoid capturing a large and frequently changing edge set (as described in §5.1), our optimization for creating and using dynamic

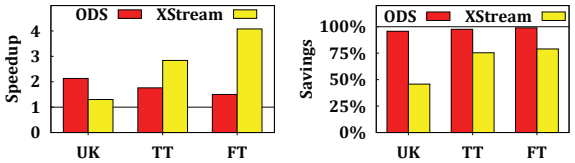


Figure 10: Speedups achieved (left) and per-iteration savings in execution time achieved (right) by **ODS** and **X-Stream** over **Baseline** using PR.

shards gets activated after a certain number of iterations (*e.g.*, 20 and 14 for TT and FT, respectively), and these (beginning) iterations do not get optimized.

Although X-Stream has better performance, the proposed optimization is still useful in practice for two main reasons. First, there are many vertex-centric systems being actively used. Our results show that the use of dynamic shards in GraphChi has significantly reduced the performance gap between edge-centricity and vertex-centricity (from 2.74× to 1.65×). Second, our performance gains are achieved only by avoiding the loading of edges that do not carry updated values and this type of inefficiency also exists in edge-centric systems. Speedups should be expected when future work optimizes edge-centric systems using mechanisms proposed in §4.3.

7 Related Work

Single-PC Disk-based Graph Processing Single-PC graph processing systems [14, 23, 31, 27, 15, 29, 10] have become popular as they do not need expensive computing resources and free developers from managing clusters and developing/maintaining distributed programs.

GraphChi [14] pioneered single-PC-based out-of-core graph processing systems. As mentioned in §1, shards are created during pre-processing and never changed during graph computation, resulting in wasteful I/O. This work exploits dynamic shards whose data can be dynamically adjustable to reduce I/O.

Efforts have been made to reduce I/O using semi-external memory and SSDs. Bishard Parallel Processor [18] aims to reduce non-sequential I/O by using separate shards to contain incoming and outgoing edges. This requires replication of all edges in the graph, leading to disk space blowup. X-Stream [23] uses an edge-centric approach in order to minimize random disk accesses. In every iteration, it streams and processes the entire unordered list of edges during the `scatter` phase and applies updates to vertices in the `gather` phase.

Using our approach, *dynamic edge-lists* can be created to reduce wasteful I/O in the `scatter` phase of X-Stream. GridGraph [31] uses partitioned vertex chunks and edge blocks as well as a dual sliding window algorithm to process graphs residing on disks. It enables selective scheduling by eliminating processing of edge blocks for which vertices in the corresponding chunks are not scheduled. However, the two-level partitioning is still done statically. Conceptually, making partitions dynamic would provide additional benefit over the 2-level partitioning.

FlashGraph [29] is a semi-external memory graph engine that stores vertex states in memory and edge-lists on SSDs. It is built based on the assumption that all vertices can be held in memory and a high-speed user-space file system for SSD arrays is available to merge I/O requests to page requests. TurboGraph [10] is an out-of-core computation engine for graph database to process graphs using SSDs. Since TurboGraph uses an adjacency list based representation, algorithms need to be expressed as sparse matrix-vector multiplication, which has a limited applicability because certain algorithms such as triangle counting cannot be expressed in this manner. Work from [21] uses an asynchronous approach to execute graph traversal algorithms with semi-external memory. It utilizes in-memory prioritized visitor queues to execute algorithms like breadth-first search and shortest paths.

Shared Memory and Distributed Graph Systems Google's Pregel [17] provides a synchronous vertex centric framework for large scale graph processing. GraphLab [16] is a framework for distributed asynchronous execution of machine learning and data mining algorithms on graphs. PowerGraph [8] provides efficient distributed graph placement and computation by exploiting the structure of power-law graphs. Cyclops [5] provides a distributed immutable view, granting vertices read-only accesses to their neighbors and allowing unidirectional communication from master vertices to their replicas. GraphX [9] maps graph processing back to the dataflow framework and presents an abstraction that can be easily implemented using common dataflow operators. Chaos [22] utilizes disk space on multiple machines to scale graph processing.

Ligra [24] provides a shared memory abstraction for vertex algorithms. The abstraction is particularly suitable for graph traversal. Work from [19] presents a shared-memory implementation of graph DSLs on a generalized Galois system, which has been shown to outperform their original implementations. GRACE [26], a shared-memory system, processes graphs based on message passing and supports asynchronous execution by using stale messages. Orthogonal to these shared memory systems, this work aims to improve the I/O efficiency of disk-based graph systems. Graph reduction techniques [12] can be used to further reduce I/O by processing a small subgraph first and then feeding values to the original graph.

8 Conclusion

We present an optimization that dynamically changes the layout of a partition structure to reduce I/O for disk-based graph systems. Our experiments with GraphChi demonstrate that this optimization has significantly shortened its I/O time and improved its overall performance.

Acknowledgments

We would like to thank our shepherd Byung-Gon Chun as well as the anonymous reviewers for their valuable and thorough comments. This work is supported by NSF grants CCF-1524852 and CCF-1318103 to UC Riverside, NSF grants CNS-1321179 and CCF-1409423 to UC Irvine, and ONR grant N00014-16-1-2149 to UC Irvine.

References

[1] GraphLab Create. https://dato.com/products/create/, 2016.

[2] BACKSTROM, L., HUTTENLOCHER, D., KLEINBERG, J., AND LAN, X. Group formation in large social networks: Membership, growth, and evolution. In *KDD* (2006), pp. 44–54.

[3] BENNETT, J., AND LANNING, S. The netflix prize. In *Proceedings of KDD cup and workshop* (2007), vol. 2007, p. 35.

[4] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *WWW* (2004), pp. 595–601.

[5] CHEN, R., DING, X., WANG, P., CHEN, H., ZANG, B., AND GUAN, H. Computation and communication efficient graph processing with distributed immutable view. In *HPDC* (2014), pp. 215–226.

[6] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., GIBSON, G. A., AND XING, E. P. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC* (2014), pp. 37–48.

[7] Friendster network dataset, 2015.

[8] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI* (2012), pp. 17–30.

[9] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. GraphX: Graph processing in a distributed dataflow framework. In *OSDI* (2014), pp. 599–613.

[10] HAN, W.-S., LEE, S., PARK, K., LEE, J.-H., KIM, M.-S., KIM, J., AND YU, H. TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC. In *KDD* (2013), pp. 77–85.

[11] KANG, U., HORNG, D., AND FALOUTSOS, C. Inference of beliefs on billion-scale graphs. In *Large-scale Data Mining: Theory and Applications* (2010).

[12] KUSUM, A., VORA, K., GUPTA, R., AND NEAMTIU, I. Efficient processing of large graphs via input reduction. In *ACM HPDC* (2016).

[13] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is Twitter, a social network or a news media? In *WWW* (2010), pp. 591–600.

[14] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. GraphChi : Large-scale graph computation on just a PC. In *OSDI*, pp. 31–46.

[15] LIN, Z., KAHNG, M., SABRIN, K. M., CHAU, D. H. P., LEE, H., , AND KANG, U. MMap: Fast billion-scale graph computation on a pc via memory mapping. In *BigData* (2014), pp. 159–164.

[16] LOW, Y., BICKSON, D., GONZALEZ, J., GUESTRIN, C., KYROLA, A., AND HELLERSTEIN, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow. 5*, 8 (2012), 716–727.

[17] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., CZAJKOWSKI, G., AND INC, G. Pregel: A system for large-scale graph processing. In *SIGMOD* (2010), pp. 135–146.

[18] NAJEEBULLAH, K., KHAN, K. U., NAWAZ, W., AND LEE, Y.-K. Bishard parallel processor: A disk-based processing engine for billion-scale graphs. *Journal of Multimedia & Ubiquitous Engineering 9*, 2 (2014), 199–212.

[19] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *SOSP* (2013), pp. 456–471.

[20] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford University, 1998.

[21] PEARCE, R., GOKHALE, M., AND AMATO, N. M. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC* (2010), pp. 1–11.

[22] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *SOSP* (2015), pp. 410–424.

[23] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-Stream: Edge-centric graph processing using streaming partitions. In *SOSP* (2013), pp. 472–488.

[24] SHUN, J., AND BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP* (2013), pp. 135–146.

[25] VORA, K., KODURU, S. C., AND GUPTA, R. ASPIRE: Exploiting asynchronous parallelism in iterative algorithms using a relaxed consistency based dsm. In *OOPSLA* (2014), pp. 861–878.

[26] WANG, G., XIE, W., DEMERS, A., AND GEHRKE, J. Asynchronous large-scale graph processing made easy. In *CIDR* (2013).

[27] WANG, K., XU, G., SU, Z., AND LIU, Y. D. GraphQ: Graph query processing with abstraction refinement—programmable and budget-aware analytical queries over very large graphs on a single PC. In *USENIX ATC* (2015), pp. 387–401.

[28] Yahoo! Webscope Program. `http://webscope.sandbox.yahoo.com/`.

[29] ZHENG, D., MHEMBERE, D., BURNS, R., VOGELSTEIN, J., PRIEBE, C. E., AND SZALAY, A. S. FlashGraph: processing billion-node graphs on an array of commodity ssds. In *FAST* (2015), pp. 45–58.

[30] ZHU, X., AND GHAHRAMANI, Z. Learning from labeled and unlabeled data with label propagation. Tech. Rep. CALD-02-107, Carnegie Mellon University, 2002.

[31] ZHU, X., HAN, W., AND CHEN, W. GridGraph: Large scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX ATC* (2015), pp. 375–386.

Version Traveler: Fast and Memory-Efficient Version Switching in Graph Processing Systems

Xiaoen Ju Dan Williams† Hani Jamjoom† Kang G. Shin
University of Michigan †*IBM T.J. Watson Research Center*

Abstract

Multi-version graph processing, where each version corresponds to a snapshot of an evolving graph, is a common scenario in large-scale graph processing. Straightforward application of existing graph processing systems often yields suboptimal performance due to high version-switching cost. We present *Version Traveler (VT)*, a graph processing system featuring fast and memory-efficient version switching. VT achieves fast version switching by (*i*) representing differences among versions as *deltas* and (*ii*) constructing the next version by integrating the in-memory graph representation of the current version with the delta(s) relating the two versions. Furthermore, VT maintains high computation performance and memory compactness. Our evaluation using multi-version processing workloads with realistic datasets shows that VT outperforms PowerGraph—running 23x faster with a 15% memory overhead. VT is also superior to four multi-version processing systems, achieving up to 90% improvement when jointly considering processing time and resource consumption.

1 Introduction

Multi-version graph processing is an important and common scenario in big data analytics. In such a scenario, each version corresponds to a snapshot of an evolving graph; a graph processing system iterates over all input versions and applies a user-defined algorithm to them, one at a time. Multi-version graph processing enables the analysis of characteristics embedded across different versions. For example, computing the shortest distance between two users across multiple versions of a social network captures the varying closeness between them [26]. Computing the centrality scores of scientific researchers across multiple versions of a co-authorship graph demonstrates their evolving impact [17].

A key element in multi-version graph processing is efficient *arbitrary local version switching*. Version switching refers to the preparation of the next to-be-processed version after computation completes on the current version. Such a procedure can be arbitrary, because the sequence of to-be-processed versions cannot be predetermined by the underlying system. It is local in that the next version commonly resides in the vicinity of the current version, demonstrating version locality.

Arbitrary local version switching has not been fully addressed before, in particular, from the perspective of the entire multi-version processing workflow. Due to version unawareness, mainstream systems, such as Pregel [23] and PowerGraph [11], perform version switching by discarding the in-memory representation of the current version and loading the next version in its entirety from persistent storage. Such an approach incurs substantial version switching time. Existing multi-version processing systems [10, 17, 19, 22] expedite the version-switching procedure by graph-delta integration. Specifically, the next version is constructed by integrating the current version with the delta representing the difference between the two versions. Albeit efficient, they either lack the support for arbitrary local switching [10, 19], incur high neighbor access penalty during computation [17], or lead to high memory overhead [22].

Towards efficient support for arbitrary local version switching, a system must balance contradicting requirements among three design dimensions: *extensibility*, *compactness*, and *neighbor access efficiency*. Extensibility refers to the easiness of creating a new graph version by extending the current one. Compactness refers to the memory overhead related to version-switching support. Neighbor access efficiency refers to the speed of neighbor access by a graph computing engine.

We present *Version Traveler (VT)*, a multi-version graph processing system enabling fast arbitrary local version switching. From a holistic view, VT balances the requirements in all three dimensions of the design space. VT consists of two novel components: (*i*) a *hybrid-com-*

pressed-sparse-row (CSR) graph supporting fast delta integration while preserving compactness and neighbor access speed during graph computation, and (*ii*) a *version delta cache* that stores the deltas in an easy-to-integrate and compact format. Conceptually, the hybrid-CSR graph represents the common subgraph shared among multiple versions in the CSR format. As a result, the subgraph is compactly stored in memory and yields high neighbor access speed—both known advantages of the CSR format. Differences among versions are absorbed by a hierarchical vector-of-vectors (VoV) representation and placed in the delta cache, leading to high version-switching speed thanks to its ability to overcome CSR's lack of extensibility.

We have implemented Version Traveler inside PowerGraph [11] by augmenting its graph representation layer with VT's hybrid-CSR graph and version delta cache. Our evaluation with realistic graphs shows that VT significantly outperforms PowerGraph in multi-version processing: VT runs 23x faster with a mere 15% memory overhead. VT also outperforms designs proposed in state-of-the-art multi-version processing systems, such as log delta, bitmap delta, and multi-version-array, achieving up to 90% improvement when jointly considering processing time and resource consumption.

The contributions of this paper include:

- the formulation of the arbitrary local version switching problem in the context of multi-version graph processing,
- a method for arbitrary local version switching with a holistic view, considering neighbor access speed, version switching speed, and compactness,
- the design of Version Traveler, a graph processing system balancing the above three requirements with two novel components, and
- the demonstration of VT's superior performance compared to state-of-the-art graph processing systems via extensive evaluation.

2 Multi-Version Graph Processing

In this section, we first discuss the characteristics of multi-version graph processing workloads, followed by a discussion on its workflow. We then summarize related work, analyze the design space for efficient multi-version graph processing systems, and discuss challenges.

2.1 Workload Characteristics

In multi-version graph processing, version switching commonly demonstrates randomness and locality. Version switching is *arbitrary*, in that the next version may precede or succeed the current version in the graph evolution.[1] Such a switching sequence may be dynamic, unable to be predetermined by the graph processing system. Version switching is *local*, in that the next version commonly resides within the vicinity—in terms of similarity—of the current one in the graph evolution.

We exemplify the demand for arbitrary local version switching with three examples. First, suppose we need to identify the cause of the varying distance between two users in a social network [26]. For simplicity, assume that the distances in versions i and k are different and that the distance changes only once along the evolution from versions i to k. If, after processing version $j = \frac{i+k}{2}$, a binary-search-style exploration algorithm finds that the distance in that version remains the same as that in version k but differs from that in version i, then the algorithm would invoke another iteration of shortest distance computation for version $m = \frac{i+j}{2}$. The switching from versions j to m is arbitrary for the supporting graph processing system. In terms of locality, although the search may oscillate between versions with high dissimilarity at the beginning, the version locality increases exponentially with the progress of the execution.

Second, in interactive big data analytics, an analyst may rerun an algorithm on a graph version, after digesting the results of the previous execution. Which version should be processed next depends on the analyst's understanding of the existing results, as well as his/her domain knowledge and intuition. This leads to arbitrary version switching from the perspective of the graph processing system. As for locality, such analysis commonly follows a refinement procedure, where significant efforts are required to zoom in and conduct in-depth analysis on a cluster of versions within the vicinity of each other.

Third, in a collaborative data analytics environment, both datasets and computing power are shared among users [5]. Individual tasks—each targeting a graph version—can be combined by the processing system, leading to multi-version graph processing. Since the next request may be enqueued during the processing of the current version and may target a version preceding or succeeding the current version in the graph evolution, version switching is arbitrary. Regarding locality, independently-submitted tasks may target similar versions. Such is the case, for example, where various algorithms are employed to capture and understand trending events in an evolving social network.

2.2 Workflow

A typical multi-version graph processing workflow is divided into multiple iterations. In each iteration, an arbitrary local graph version is processed. Systems de-

[1] More broadly, in a non-linear graph evolution scenario [5], the next version may reside in a different branch than the current version.

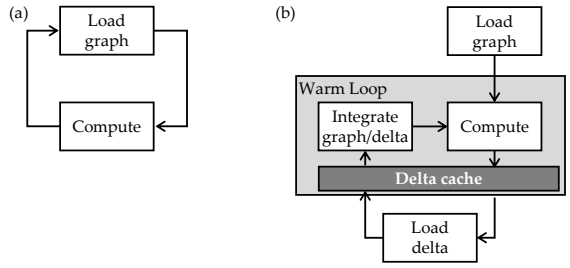


Figure 1: Version switching workflow, (a) with and (b) without the use of deltas

signed for individual graph processing tasks are unable to recognize or take advantage of the evolution relation among versions. Treating each version as a standalone graph, such systems first fully load the version from persistent storage into memory and then execute a user-defined graph algorithm over it (cf. Figure 1a).

When versions of a working set share a substantial common subgraph, working with deltas—representations of the differences between graph versions—can be more efficient. Figure 1b shows the multi-version processing workflow with deltas. After the first version is loaded and processed, switching to a subsequent version can be achieved by integrating the current version in memory with deltas relating the current and the next versions. In general, deltas are much smaller than full graphs [26, 31]. As a result, they can be cached in memory, further improving the efficiency of version switching.

2.3 Related Work on Graph/Delta Designs

We focus the discussion of related work on in-memory graph and delta representations, because they determine the efficiency of the switching loop (cf. Figure 1b).[2] For graphs, we specifically focus on representations related to neighbors of vertices, because they differentiate graphs from regular table-form datasets.

As a result, we exclude other active research directions, such as programming paradigms [11, 14, 15, 20, 23, 24, 30, 32], out-of-core processing [19, 27], load balancing [16], failure recovery [28], streaming [10], dataflow-based processing [9, 12], and performance evaluation [21]. We also exclude work within the broad scope of multi-version processing but not dedicated to in-memory graph/delta design in the context of arbitrary local version switching. Examples are streaming processing [10, 19],[3] parallel multi-version processing [13], multi-version algorithm design framework [26], and multi-version dataset management [5, 6]. [4]

We study related work by asking three questions:

- Does it provide high computation performance? In particular, does it support fast access of the neighbors of a vertex?[5]
- Does it support fast version switching?
- Does it store graphs and deltas compactly?

Graph. We study two common graph representations: compressed sparse row (CSR), adopted in PowerGraph [11] and GraphX [12], and a vector-of-vectors (VoV) design, adopted in Giraph [1].

In CSR (cf. Figures 2b and 2c), all neighbors are packed in an array. A pointer array maintains the address of the first neighbor of each vertex. The set of neighbors for vertex i is thus marked by the values of vertices i and $i+1$ in the pointer array. This representation enables fast access to a vertex's neighbors. Version switching is slow, however. This is because modifying a vertex's neighbor affects pointers and neighbors of all vertices following the one being modified.

As for VoV (cf. Figures 2d and 2e), the first-level vector functions as the pointer array in CSR, locating the neighbors of a vertex according to the vertex id. Each second-level vector represents the neighbors of a vertex. This format also supports fast neighbor access. In addition, the neighbors of a vertex can be modified without affecting other vertices, thus enabling fast version switching. Its shortcoming is the memory overhead due to maintaining auxiliary information, such as the start and end positions of each vertex's neighbors.[6]

Delta. Previous work has used a compact log-format structure to represent deltas in streaming processing [10]. A log delta consists of an array of log entries, each specifying an edge via its source and destination vertex ids (and an optional edge id) and whether the edge should be added or removed (i.e., an opcode). Log deltas are

[2] Both graphs and deltas may have different representations in memory and on disk. We focus on in-memory representations, due to their significance in the warm loop of the version switching workflow.

[3] Streaming processing is a special case of multi-version graph processing, where version switching is always forward and versions are only processed once. They are insufficient for the general multi-version scenario, where switching is arbitrary and a cached version is repeatedly accessed by multiple algorithms.

[4] Such work investigates the tradeoff between storage space and recreation speed of a dataset version, focusing on organizing versions on disk instead of in-memory data structure optimization.

[5] In this paper, we equate computation performance with neighbor access efficiency for two reasons. First, computation related to graph algorithms affects all systems in the same way and is out of scope. Second, in the computation stage, a system supports neighbor access and vertex/edge data access. Assuming the storage of data in sequence containers and their identical impact on all systems, computation performance is determined only by neighbor access efficiency.

[6] Such overhead is non-trivial. For example, a 24-byte per-vector overhead (cf. Figure 2d, "start," "end of contents," and "end of storage" pointers each consume 8 bytes) amounts to a 40% overhead for representing the entire out-neighbor array for the Amazon08 dataset [7, 8], assuming 4-byte vertex/edge ids.

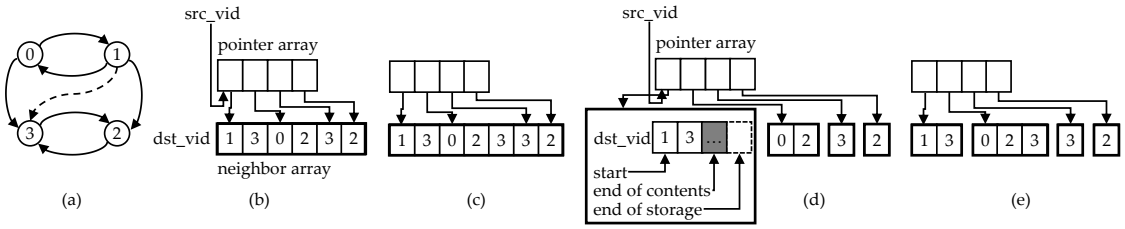


Figure 2: Graph representations: (a) illustrates two versions of a graph. A circle represents a vertex (vertex id inside) and an arrow represents an edge (edge id omitted). The first version consists of solid-arrow edges. The second version has one more edge (illustrated by a dashed arrow). (b) and (c) demonstrate the CSR representation of the out-edges of the two versions. (d) and (e) demonstrate the vector of vectors format. For clarity, each element in the neighbor array in (b)–(e) shows only the destination vertex id and omits the edge id.

compact and have no negative impact on the neighbor access efficiency during graph processing. This is because log deltas are conceptually applicable to all graph representations as-is. During graph-delta integration, log deltas are fully absorbed in the graph version. The cost of graph-delta integration is high, however, because all log entries in deltas relating the current and the next versions need to be applied during version switching.

Alternatively, a system could co-design graph and delta representations to minimize the integration cost. For example, affected neighbor vectors of a VoV graph may be copied and updated in a delta, reducing version switching to simple and fast vector pointer updates but losing compactness. LLAMA [22] partially mitigates the compactness loss by separating modified neighbors related to a version into a dedicated consecutive area in the neighbor array, avoiding copying unmodified neighbors. CSR's pointer array is transformed to a two-level translation table. The first level consists of per-version indirection tables, each bookkeeping a set of second-level pages associated with a version. A second-level page contains a series of vertex records—equivalent to a fragment of CSR's pointer array—with each record indicating the start of a vertex's neighbors.[7] LLAMA's version switching incurs nearly zero time cost: conceptually, only an indirection table pointer needs to be updated. Its use of page-level copy-on-write for the second-level pages holding vertex records, nevertheless, requires the copy of an entire page even if only one vertex in the page has a modified neighborhood, hindering its compactness.

GraphPool [17] maintains the union of edges across all versions in the graph. Its deltas are per-version bitmaps over the graph's edge array, where a bitmap's n-th bit indicates the existence of the corresponding edge in that version. Version switching is simple: a bitmap pointer is adjusted to point to the next version. In the computation stage, however, this approach requires bitmap checking for determining whether an edge exists in the current version, incurring neighbor access penalty.

[7]Neighbors belonging to the same vertex but stored in separate areas—each containing per-version modifications—are concatenated via *continuation records* such that only one start position needs to be maintained for a vertex's neighbors per version.

2.4 Design Dimensions and Challenges

Summarizing lessons learned from related work, we point out that the design of graph and delta must balance between three dimensions: extensibility, compactness, and neighbor access efficiency.

Extensibility. Efficient version switching requires that a delta be easily integrated with a graph. From a data structure perspective, it requires that the neighbors of a vertex be easily extended to reflect the evolution from one version to another. This, in turn, requires that either the data structure representing the neighbors of a vertex support efficient modification (i.e., insertion and removal) or the collection of the neighbors of a version be easily replaced by that of another version.

Compactness. Compact graph and delta representations enable caching a large number of versions, leading to low delta cache miss rate and high version switching efficiency. Moreover, real-world large graphs commonly have millions of vertices and millions or billions of edges, making the compactness of the neighborhood data structure a primary requirement.

Access Efficiency. A common and crucial operation during computation is to access a complete collection of neighbors for a vertex. Fast neighbor access requires limiting the number of lookups in the integrated graph/delta data structure. Ideally, only one lookup is sufficient for locating the first neighbor of a vertex. The remaining neighbors can then be accessed sequentially.

The main design challenge is to carefully balance the requirements from the above three dimensions and co-design graph and delta representations such that they are extensible, compact, and efficient in neighbor access. Achieving the balance is difficult, as witnessed by exist-

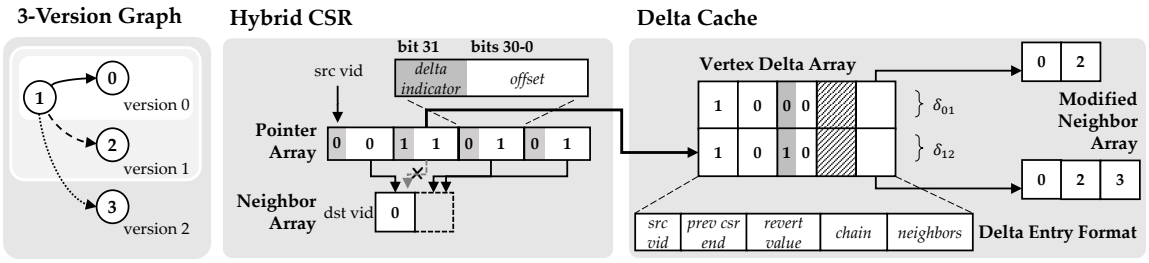


Figure 3: Hybrid graph representation

ing designs, because those requirements commonly lead to contradicting design choices.

3 Version Traveler

We introduce Version Traveler (VT), a graph processing system that features a graph/delta co-design achieving compactness, extensibility, and access efficiency. Residing in the core of VT are two innovative components—a hybrid graph and a hybrid delta cache—bringing together fast neighbor access and compactness of CSR and high extensibility of VoV.

3.1 Hybrid Graph

VT's hybrid graph augments CSR in a way that achieves extensibility while remaining compact and efficient in neighbor access (cf. Figure 3). It avoids costly in-place modification to CSR's neighbor array by storing vertices with a modified neighborhood in a version delta cache.

CSR's neighbor array is created during the loading of the first version—also referred to as the root version—and then remains constant. Each subsequent version is loaded into the delta cache, in the form of a series of *vertex delta entries*. Each entry contains information related to the updated neighbors of a vertex, as well as metadata to support neighbor access and version switching (cf. Figure 3). VT reserves a *delta indicator bit* in each entry of CSR's pointer array to indicate the placement of a vertex's neighbors for the current version: in CSR's neighbor array or in the vertex delta cache.

Neighbor Access. In a conventional CSR, neighbors of vertex *vid* are bounded by the pointers of vertices *vid* and $vid+1$. For VT's hybrid CSR, neighbor access may be directed to either CSR's neighbor array or the *neighbors* field of a delta entry, depending on whether the neighbors are stored (cf. `access_neighbors` in Algorithm 1). Each delta entry maintains the end position of the preceding vertex's neighbors in CSR's neighbor array (in the *prev_csr_end* field), such that the end position of *vid* can be determined regardless of whether neighbors of $vid+1$ are stored in CSR's neighbor array or the delta cache (cf. lines 4–7).

Algorithm 1 Neighbor Access and Delta Application

```
1:  procedure ACCESS_NEIGHBORS(vid)
2:      if csr_ptrs[vid].in_delta = true then
3:          return cache[csr_ptrs[vid]].nbrs
4:      else if csr_ptrs[vid+1].in_delta = false then
5:          return csr_nbrs[csr_ptrs[vid], csr_ptrs[vid+1]]
6:      else return csr_nbrs[csr_ptrs[vid],
7:                  cache[csr_ptrs[vid+1]].prev_csr_end]
8:  procedure DELTA_APPLICATION(δ_ij, opcode)
9:      for e in δ_ij do
10:         if opcode = apply then                    ▷ apply δ_ij
11:             csr_ptrs[e.src_vid] ← offset(e)
12:             csr_ptrs[e.src_vid].in_delta ← true
13:         else csr_ptrs[e.src_vid] ← e.revert_value  ▷ revert δ_ij
```

Delta Application and Reversion. VT performs version switching by applying or reverting deltas (cf. `delta_application` in Algorithm 1). When applying δ_{ij} to switch from versions i to j, VT iterates over entries belonging to δ_{ij} in the delta cache and, for each entry, updates the corresponding entry (according to the *src_vid* field) in the CSR pointer array with the delta entry's offset in the delta array and sets the delta indicator bit. Reverting δ_{ij} consists of restoring the *revert value* field—which contains the saved value for version i's CSR pointer entry—to the corresponding entry in the CSR pointer array for each entry in δ_{ij}.

Example. We use a 3-version graph in Figure 3 to illustrate neighbor access and version switching. The hybrid CSR in Figure 3 represents the state of version 2. The three out-neighbors of vertex 1 can be accessed from its delta entry (δ_{12}). For vertex 0, neighbor access requires obtaining the start position from its CSR pointer, due to its cleared delta indicator bit, and the end position from *prev_csr_end* of vertex 1's delta entry. The difference between the two is 0 ($0-0=0$), indicating that vertex 0 has no out-neighbors. In order to switch from versions 2 back to 1, VT reverts δ_{12}, which has only one entry related to vertex 1. Its *revert value* field, of which the delta indicator bit is set and the offset is 0, is restored to vertex 1's CSR pointer entry. After reversion, vertex

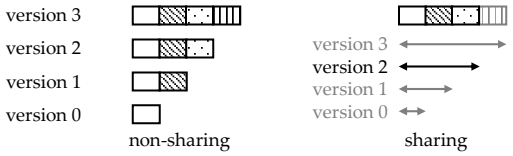


Figure 4: Illustration of the concept of Sharing

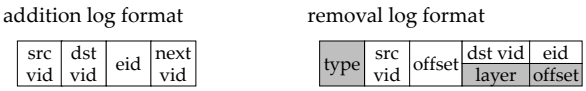


Figure 5: Delta log format

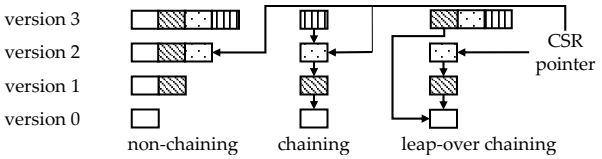


Figure 6: Illustration of the concept of Chaining

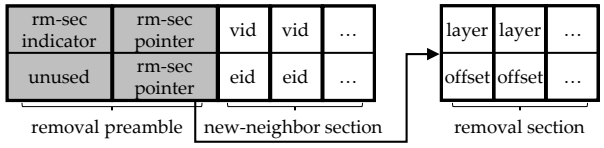


Figure 7: Neighbor vector format

1's CSR pointer entry will point to the first entry in the delta array, which corresponds to vertex 1's delta entry for version 1.

3.2 Hybrid Delta

For simplicity, in Section 3.1, we assume that a delta entry maintains the entire neighbors of a vertex (cf. Figure 3). This is memory-inefficient for vertices with a large number of neighbors and small amount of per-version modifications, due to numerous redundant copies of neighbors. To improve compactness, we propose two complementary solutions: *Sharing* and *Chaining*. Sharing preserves access efficiency and trades extensibility for compactness. Chaining preserves extensibility and trades access efficiency for compactness.

3.2.1 Sharing

Concept. Sharing reduces the memory footprint by merging a vertex's delta entries spanning multiple versions into one shared entry. Figure 4 shows an example with four versions of a vertex, each adding one neighbor to its base. When they share a delta entry, there exists only one neighbor vector, containing the neighbors of the vertex related to the current version being processed. The challenge is to compactly specify how the shared vector is modified during version switching. VT maintains this information in addition and removal *delta logs*.

Delta Representation. In the Sharing mode, VT does not create delta cache entries with copies of modified neighbor arrays. Instead, it creates log entries: specifying the neighbors it would have added to or removed from the neighbor array in an addition or removal log. Each entry in the addition log array (cf. Figure 5) contains the source and target vids of the added edge, as well as the edge id. Logs associated with the same vertex (the source vid in the out-neighbor case) are continuously stored. A *next vid* field indicates the start position of the logs associated with a subsequent vertex. The format of a removal log entry—gray fields apart—is similar. Its *offset* field refers to the offset within the neighbor array where the removal should take place.

Neighbor Access. Sharing has no effect on neighbor access. When multiple versions of a vertex share a neighbor vector, the CSR pointer header points to the same delta entry containing the vector for all versions.

Delta Application/Reversion. In the Sharing mode, version switching resorts to log-based delta application/reversion, similar to streaming processing [10, 19]. During delta application, for an neighbor addition, the neighbor is simply appended to the end of the neighbor vector. For a removal, VT removes the neighbor at the offset according to the *offset* value in the removal log entry. Delta reversion follows the inverse procedures.

3.2.2 Chaining

Concept. Chaining refers to the representation of a vertex's neighbors with a chain of vectors, each containing a subset of neighbors and capturing the difference between the version associated with it and its base version. In Figure 6, with each version chained onto its base, only one neighbor needs to be maintained per version. Redundant copies are eliminated, improving compactness. Extensibility remains the same: to switch from versions 1 to 2, for example, we need to adjust only the CSR pointer to version 2's delta entry, regardless of whether that entry's neighbor vector is chained onto another. Access efficiency decreases in Chaining, because of the need to switch among multiple neighbor vectors. Chaining imposes two new challenges to delta design: chaining beyond the base, called *Leap-Over Chaining*, and removal from ancestors, called *indirect removal*.

Delta Representation. Leap-Over Chaining intends to accelerate neighbor access. In Figure 6, with each ver-

sion chained onto its base, the neighbor access for later versions leads to considerable performance hit, limiting Chaining to a small set of adjacent versions in the graph evolution relation. Leap-Over Chaining enables the chaining of a delta entry on an indirect ancestor version. For instance, version 3 in Figure 6 can be chained onto version 0.

To support Chaining, in particular Leap-Over Chaining, we introduce a *chaining* field to the delta entry format (cf. Figure 3). When Chaining is disabled,[8] the entry is a standalone entry with a complete copy of neighbors. When Chaining is in use, VT saves a pointer to the entry upon which the current one is based along the chain to the latter's *chaining* field. Similar to CSR pointers, a chaining pointer uses its most significant bit to indicate whether the offset is for CSR or for the delta array.

To support indirect removal, a neighbor vector is divided into two sections: a new-neighbor section and a removal section (cf. Figure 7). An element in the new-neighbor section represents a new neighbor added to the vertex in the current version. An element in the removal section corresponds to a removed element, with *layer* indicating the neighbor vector in the chain where the removal should take place and *offset* the position of the to-be-removed neighbor within the vector. To improve compactness, we overload the first element in the new-neighbor section: it is marked with a special flag if the removal section exists, in which case the second element contains a pointer to the removal section;[9] otherwise it contains the first added neighbor.

Effect on Removal Log. Since Chaining introduces the separation of new-neighbor and removal sections, Sharing's removal log format needs to be adjusted (cf. gray fields in Figure 5). Specifically, a *type* field is added to differentiate the two sections. A {layer, offset} pair in a removal section is stored similarly to a {vid, eid} pair in a new-neighbor section. During delta application, if a removal takes place in the current vertex's new-neighbor section, then the corresponding {vid, eid} pair is removed. Otherwise, the {layer, offset} pair is inserted to the current vertex's removal section. The inverse procedure achieves delta reversion.

Neighbor Access. Given a vertex, VT first locates its entry via the CSR pointer header. If the neighbors are stored in a delta array entry whose Chaining mode is turned on, then VT iteratively accesses neighbors stored in entries along the chain, skipping neighbors that no longer exist in the current version using the removal sections. Otherwise, it follows the common neighbor access procedure, as described in Section 3.1.

Delta Application/Reversion. Except for Chaining's effect on removal logs, both delta application and reversion follow the description in Section 3.1.

[8] That is, setting all bits in *chaining* to one.

[9] The first two elements are also referred to as *removal preamble*.

3.2.3 Relationship between Chaining and Sharing

Chaining and Sharing are similar in that they both aim at reducing memory consumption by storing only the difference among versions. Sharing is a good choice when the size of neighbors is large and the delta size is small. A large neighborhood leads to a considerable gain in compactness over a full-neighbor-copy approach, whereas a small delta entails a moderate cost for log-based version switching. Chaining is useful when both the sizes of neighbors and delta are large. Similar to Sharing, a large neighbor size leads to a substantial gain in compactness for Chaining. A large delta entails Chaining's superiority to Sharing, due to the avoidance of the latter's costly log-based version switching procedure.

Another way to compare the two is when the concatenation of neighbors occurs. Chaining performs the concatenation in a chain at the computation stage. Sharing performs the concatenation at the version switching stage. Due to the different delta formats used in Chaining and Sharing, the concatenation in Chaining is lighter-weight than that in Sharing. As for the number of concatenation performed for a vertex, the concatenation in Chaining needs to take place when a vertex's neighbors are accessed. The cost of concatenation in Chaining is thus magnified if a vertex is iteratively processed by an algorithm. The concatenation in Sharing is, in contrast, guaranteed to be once per vertex per version switching.

VT supports Sharing and Chaining as operation modes, complementing the default full-neighbor-copy mode (referred as Full mode). It enables them when the estimated cost of version switching and the potential impact on the computation stage are justified by the amount of memory saving. The current VT implementation supports flexible threshold-based policies: when creating a new delta for a vertex, VT feeds the number of neighbors in its base version and the current delta size related to that vertex to a configurable policy arbitrator function, which determines the activation of Sharing or Chaining.

3.3 Implementation

We implement VT by integrating it with PowerGraph [11], replacing the latter's graph representation with VT's hybrid CSR graph and delta/log arrays. VT operates seamlessly with PowerGraph's computation engine layer, thanks to its full support of the same computation-stage graph abstraction viewed from an engine. This also demonstrates VT's broad applicability to existing graph processing systems.

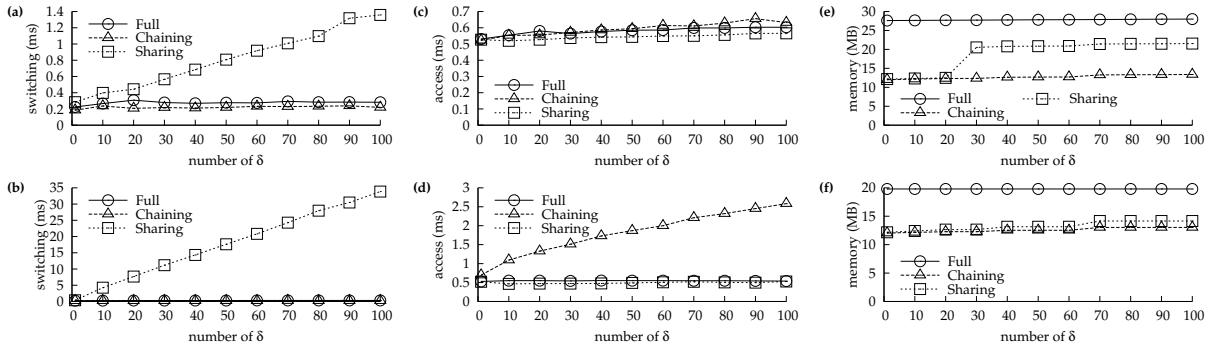


Figure 8: Microbenchmark results (top row for additions and bottom row for removals)

4 Evaluation

We first demonstrate the three-way tradeoff among extensibility, compactness, and access efficiency, showing the relative advantage of Full, Chaining, and Sharing. We then compare the performance of VT against PowerGraph and several multi-version reference designs.

4.1 Microbenchmark

Design. The goal of microbenchmarking is to evaluate the relative effectiveness of Full, Sharing, and Chaining in balancing the three-way tradeoff. Since the overall tradeoff on a graph is the accumulative effect of the same tradeoff on each vertex, we conduct microbenchmarking from a vertex's perspective. Trends in the microbenchmark results are applicable to varying graph sizes, given the accumulative nature of the per-vertex tradeoff.

We construct a graph with 1000 identical high-degree stars. For each star, only the *center vertex* has a non-empty set of out-neighbors—default to 1000. Each center vertex thus provides the opportunity for an in-depth study of the per-vertex tradeoff. To evaluate it in a multi-version scenario, we create two versions: a source version and a target version. The target version differs from the source by randomly adding or removing out-neighbors of center vertices.

Two key factors related to a delta are its size and the ratio of additions to removals. Prior work shows that the difference between consecutive versions is commonly within 1% of the graph size [26]. For each star, given the default 1000 edges in the base version, we vary the delta size from 1 to 100, corresponding to 0.1% to 10% of the size of the star. The total vertices and edges in the graph thus vary between 0.9 to 1.1 million. We also fix the operation types in a delta: a delta consists of either edge additions or edge removals.

We evaluate extensibility by measuring the version switching time from the source version to the target version, neighbor access efficiency by measuring the time for iterating through all the out-neighbors of center vertices in the target version, and compactness by measuring the memory used for maintaining the graph connectivity information of both versions. All measurements are conducted on a host with 8 3GHz vCPUs and 60GB memory.

Version Switching. Figures 8a and 8b compare the version switching performance. The performance of Full and Chaining is comparable and remains constant, regardless of the edge modification types in deltas or the delta size, because both approaches require adjusting only CSR pointer values for center vertices. The cost of Sharing linearly grows with the delta size, due to the need to parse a log array whose size is proportional to the delta size. Comparing edge additions with removals, the cost of the former is significantly lower than the latter. This is because, additions translate to appending neighbor records to the end of the neighbor vector and removals involve data movement within the vector.

Access. Figures 8c and 8d compare the neighbor access speed. Full and Sharing perform equally well for both additions and removals. Since the cost of neighbor access is proportional to the neighborhood size, it linearly increases and decreases with the size of delta in the cases of addition and removal, respectively. Chaining leads to the worst performance in both cases, due to its cost of indirection. The cost is moderate in the case of edge additions, because there is one and only one indirection during neighbor access—that is, the switching from the newly added neighbors to the existing ones—regardless of the delta size. The cost of indirection becomes significant for edge removals, because each removal separates a previously continuous neighbor range into two, introducing one more indirection during neighbor access. The cost thus linearly grows with the delta size.

Memory. Figures 8e and 8f show the memory footprint. In both addition and removal cases, Chaining and Sharing lead to significant memory savings comparing with

Table 1: Graphs, algorithms, and reference designs

dataset	V (M)	E (M)	description
Amazon08	0.7	5.2	similarity among books
Dblp11	1.0	6.7	scientific collaboration
Wiki13	4.2	101.4	English Wikipedia
Livejournal	5.4	79.0	friendship in LiveJournal social network
Twitter	41.7	1468.4	Twitter follower graph
Facebook	0.1	1.6	friendship in regional Facebook network
GitHub	1.0	5.7	collaboration in software development

algorithm	description
nop	access neighbor and return
bipart	max matching in a bipartite graph
cc	identify connected components
PageRank	compute rank of each vertex
sssp	single-source shortest path
tc	triangle count

ref. design	description
csr	use CSR graph and log delta
log	use VoV graph and log delta
bitmap	maintain union of neighbors in all versions in VoV graph and use bitmap delta
m-array	use multi-version-array graph/delta

Full. Intuitively, the cost of Full linearly grows with the delta size in the addition case and linearly decreases in the removal case. Our measurements, however, show mostly constant memory footprints in both cases, due to (1) the capacity doubling effect and (2) no capacity reduction upon removal in the vector implementation in our testbed (glibc 2.15). For Chaining and Sharing, the memory footprint grows with the size of delta, regardless of the type of edge modifications. This is because, for both additions and removals, Chaining needs to maintain the modifications either in the neighbor vector (for additions) or in the removal section (for removals). Similarly, Sharing maintains the modifications in the log arrays.

4.2 Macrobenchmark

Reference Designs. We compare VT with PowerGraph [11]—a high-performance system targeting individual graph processing—and four reference multi-version processing system designs (cf. Table 1) reflecting different combinations of graph and delta formats. Specifically, we evaluate CSR+log, VoV+log, VoV+bitmap, and multi-version-array. They mirror design choices made in PowerGraph, Giraph [1], GraphPool [17], and LLAMA [22], and are abbreviated to *csr*, *log*, *bitmap*, and *m-array*, respectively.

Workloads. Table 1 summarizes the datasets and algorithms. The Facebook [31] and GitHub graphs are collected as dynamically evolving graphs. The remaining five graphs are collected as static graphs [7, 8], for which deltas need to be created. Since deltas among consecutive versions are commonly within 1% of the graph size [26], we vary the delta size from 0.01% to 1%. We select $\delta = 0.1\%$ as a middle ground and show most of the evaluation results with this configuration. The total number of cached versions n varies broadly from 1 to 100. Unless otherwise specified, we use uniform add-only deltas: each delta consists of edge additions uniformly distributed over a graph. Graphs evolve linearly: version i is created by iteratively applying $\delta_{j,j+1}, j = 0 \ldots i-1$ to the root version (i.e., version 0). Version switching is local, in that all versions are within the range of $n\delta$ from the root version. Version switching is arbitrary. That is, the next version j is selected independently to the current version i, may precede or succeed i, and do not need to be consecutive to i.

Since machines with large memory and many cores become popular and affordable [25, 29], VT's evaluation focuses on single-host setting. The elimination of inter-host communication cost in graph processing stage further highlights the effect of neighbor access efficiency. All measurements except those related to the Twitter graph [18] are performed on a host with 8 3GHz vCPUs and 60GB memory. Twitter-related workloads run on a host with 32 2.5GHz vCPUs and 244GB memory.

Metrics. The requirements on extensibility, compactness, and access efficiency naturally lead to the use of time and memory consumption as two basic metrics. In addition, inspired by the resource-as-a-service model in the economics of cloud computing [4], we introduce a penalty function as a third metric: $p = (t_s + t_c)^\alpha \times m^\beta$. The penalty p is a function of the version switching time t_s, the computation time t_c, and memory consumption m. α and β are weights associated with time and memory resource. If the per-time-unit monetary cost is determined only by memory consumption, then assigning 1 to both parameters equates the penalty with the per-task monetary cost. We use $\alpha = 1$ and $\beta = 1$ in our evaluation. When appropriate, we report penalty score p in the form of *utility improvement*: the improvement of VT over a reference system *ref* is calculated as $\frac{p_{ref} - p_{vt}}{p_{ref}}$.

Delta Preparation. For each system/workload setting, deltas corresponding to versions accessed in that workload are populated in memory, according to the delta design employed by that system, before the start of the workload. For VT, we employ a threshold-based policy (cf. Section 3.2.3), determining the delta format according to the number of neighbors in its source version and switching from Full to Sharing to Chaining as the number increases. We sample the threshold space for Full-Sharing and Sharing-Chaining transitions and report the

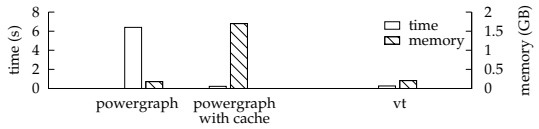


Figure 9: Comparison of VT and PowerGraph

lowest penalty score.

Comparison with PowerGraph. We evaluate the performance of PowerGraph by running SSSP on Amazon08 with ten 0.1% δs in two scenarios. First, we measure the performance of PowerGraph as-is, with a graph version loaded from persistent storage in its entirety at the beginning of each task. Second, we augment PowerGraph with full-version caching, storing each version in the working set as a full graph copy in memory.

Figure 9 shows that VT significantly outperforms PowerGraph in both scenarios. VT's processing speed is on a par with that of PowerGraph with full-graph caching and is 23x faster than that of PowerGraph without caching. This is due mainly to PowerGraph's substantial loading time when caching is disabled. VT's memory footprint is close to that of PowerGraph without caching (incurring a 15% overhead) and is only 12% of that of PowerGraph with full-graph caching—a 7.3x enhancement. Overall, VT improves utility by 86% and 95% over PowerGraph with and without caching, respectively.

Comparison with Multi-Version Designs. Figure 10 compares four multi-version designs with VT, executing nop on Amazon08 with ten 0.1% δs. *Csr* incurs prohibitive switching cost, due to CSR's low extensibility. It, nevertheless, yields the highest performance and has the smallest memory footprint. Both *log* and *bitmap* consume more memory than VT. *Bitmap* incurs a computation-stage penalty due to bitmap checking. *Log*'s switching cost is 7.4x that of VT.

Regarding *m-array*, its neighbor access time and version switching time are significantly shorter than the other designs. Its memory consumption, however, is much higher than the other. It is important to note that *m-array*'s superior performance is an outcome of efficient *implementation* of LLAMA, not a result of the multi-version-array design. This is because, after a version becomes ready for processing, all things being equal, *csr* should yield the highest neighbor access performance for the nop workload. The difference between *m-array* and *csr* is then due to the framework-related overhead: *m-array* is measured with LLAMA and *csr*—as well as the other designs—is measured with PowerGraph-based implementation. Had we ported *m-array* to PowerGraph, its performance would be at best on a par with *csr*, and thus also close to VT.

M-array's high memory consumption is a result of the multi-version-array design. For Amazon08 with 0.1% δs, each version contains 5.2K new edges. Uniformly distributed, those edges affect 5.2K vertices' neighborhood. In LLAMA, with a 16-byte vertex record[10] and a 4KB page, the entire vertex record array for the root version spans 2.7K pages, which is also the expected number of pages affected when the 5.2K vertices with modified neighborhood are uniformly distributed. This yields a 100% memory overhead in terms of per-version vertex record array—because the entire 2.7K pages containing the root multi-version array need to be copied for each version—and a 21.5% overhead when the entire graph connectivity structure (with neighbor arrays) is considered. Such an overhead is prohibitively expensive for large graphs. In contrast, VT has a smaller footprint for the root version and, more importantly, incurs only a 0.6% per-version overhead for the graph connectivity structure in its Chaining mode.

Figure 10 confirms our expectation on the advantages and shortcomings of existing designs. Given *csr*'s low extensibility and *m-array*'s high memory consumption, we focus on comparing VT with *log* and *bitmap* for the rest of the evaluation.

Comparison with *log* and *bitmap*. Figure 11 summarizes the results comparing VT with *log* and *bitmap*, each with 10 δs of size 0.1%. VT consistently outperforms both systems in all but one case. Except the Twitter-SSSP workload, VT runs 2–17% faster in average per-version processing time and achieves 17–34% memory saving and 19–40% utility improvement.

Running SSSP over the Twitter graph, VT runs 88% faster than *log* but 19% slower than *bitmap*. This is because, given the size of the dataset, the configuration of the supporting hardware, and the characteristics of the algorithm, the difference in version switching dominates the overall processing efficiency. *Log* falls far behind VT, due to the former's need of log replaying during version switching. VT's delta application, although efficient and highly parallelized, is still a heavier-weight operation compared to *bitmap*. Combining time and memory consumption, the net effect is that VT outperforms *log* by 90% and is on a par with *bitmap* in utility improvement.

Varying Deltas. We compare VT with *log* and *bitmap* by executing SSSP on Amazon08, varying the size of delta from 0.01% to 1% and the number of deltas from 10 to 100. Fixing the delta size to 0.1% and varying the number of deltas from 10 to 100, we observe that VT's utility gain remains high with respect to *log* and *bitmap* (cf. Figure 12a). Compared to *log*, VT's memory saving

[10] A vertex record consists of version id, offset into the neighbor array, number of new edges for the current version, and an optional out-degree, each occupying 4 bytes.

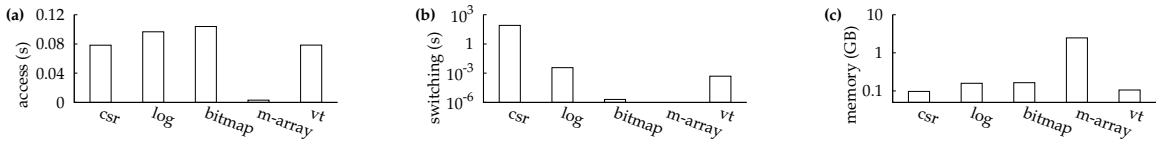


Figure 10: Comparison with existing graph/delta designs

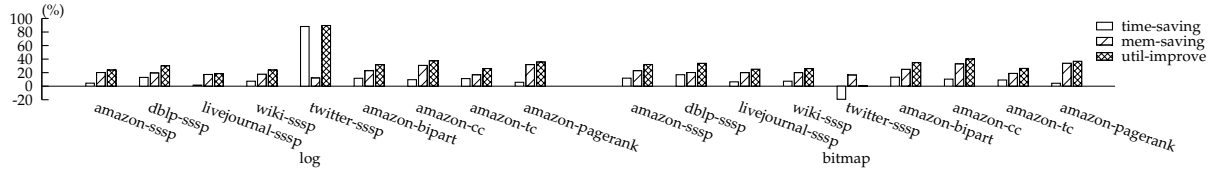


Figure 11: Comparison of VT with *log* and *bitmap* across all datasets and algorithms with 10 0.1% δs

reduces with the increasing number of deltas, because the memory consumption of the delta cache grows with the number of deltas, gradually neutralizing the benefit of the use of hybrid CSR. VT's gain due to the reduction of version switching time increases with the number of deltas, however. Overall, with these opposite trends, VT's utility gain remains high. Compared to *bitmap*, VT's memory saving remains high, because of *bitmap*'s need to maintain per-version bitmaps. VT's saving in processing time reduces, however, because the impact of *bitmap*'s saving in version switching time increases with the number of deltas, compensating for *bitmap*'s neighbor-access slowdown in the computation stage. The overall effect of these opposite trends is VT's constantly high utility gain with respect to *bitmap* across a wide range of versions.

Fixing the number of deltas to 10 and varying the size of delta from 0.01% to 1%, we observe that VT's utility gain gradually reduces (cf. Figure 12b). Compared to *log*, VT's gain peaks at $\delta = 0.01\%$, thanks to its efficient graph-delta representation. VT's gains for $\delta = 0.1\%$ and $\delta = 1\%$ are similar: larger deltas reduce VT's advantage in memory representation but amplify its reduction of version switching cost. Compared to *bitmap*, VT's utility gain remains high for $\delta = 0.01\%$ and $\delta = 0.1\%$, but drops significantly for $\delta = 1\%$. Note that, the maximum distances among versions—in terms of dissimilarity—are the same for 100 0.1% δ (in Figure 12a) and 10 1% δ (in Figure 12b). Yet, VT's gain with respect to *bitmap* is much higher in the former case. This is because VT's memory saving is more significant when *bitmap* needs to maintain a larger number of per-version bitmaps in order to track the neighbor-version relation.

Skewed and Add/Remove Workloads. Figure 13 compares VT's performance across three types of workloads, all with ten 0.1% δs. The first is a uniformly distributed add-only delta type, same as those used throughout the evaluation. The second is a skewed add-only delta, in which the probability of adding a new edge to a vertex is proportional to the latter's degree in the root version. The third is a mixed add/remove delta type, with each delta maintaining a removals/total operations ratio varying from 0.1% to 10%. VT consistently outperforms *log* and *bitmap* in all the three workloads.

Effectiveness of Optimization. Figure 14 summarizes the effectiveness of Sharing and Chaining. Reusing the workload of SSSP-Amazon with ten 0.1% δs, we first enforce a fixed delta format, measuring the performance of Full, Sharing, and Chaining individually. We then combine Full and one of the two optimization approaches and report the minimum achievable penalty. All results are then normalized to those of VT. The effectiveness of Sharing and Chaining is demonstrated by the superiority (in terms of penalty) of a combined delta preparation strategy (e.g., Full-Sharing) to both approaches when applied individually (e.g., Full and Sharing). It is also demonstrated by VT's superiority—with all three delta formats combined—to the five alternatives.

Realistic Evolving Workloads. We compare VT with *log* and *bitmap*, using two 10-version graphs generated from the evolving Facebook friendship and GitHub collaboration graphs,[11] respectively. Figure 15 shows their evolution trends. Specifically, we choose 10 consecutive days towards the end of the collected periods for the two graphs[12] and combine newly established friendship/collaboration relations in each day into a delta. Friendship/collaboration relations existing before that 10-day

[11] In the GitHub graph, the collaboration (i.e., edge) between two users (i.e., vertices) is established when they start to work on at least one shared repository. The initial state of the graph is set to empty. Its evolution spans between March 2011 and July 2015. We generate this graph via the use of GitHub API [2] and GitHub Archive [3].

[12] For the Facebook graph, daily delta size reduces drastically towards the end of the collected period, which might be caused by limitations of the collection method. We avoid those anomalies when creating the multi-version graph for evaluation.

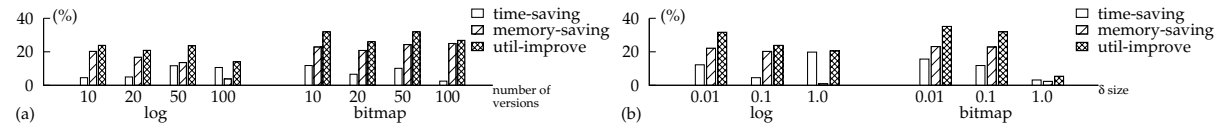


Figure 12: Varying number of δs and δ size. (a) Fixing $\delta = 0.1\%$ and varying number of versions between 10 and 100. (b) Fixing number of versions to 10 and varying δ between 0.01% and 1.0%.

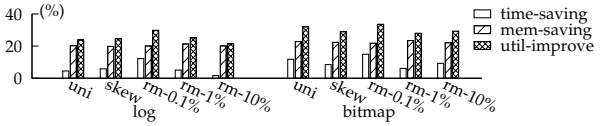


Figure 13: Uniform, skewed, and add/rm deltas

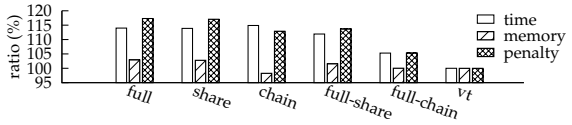


Figure 14: Effectiveness of optimization, with 10 0.1% δs (normalized to VT)

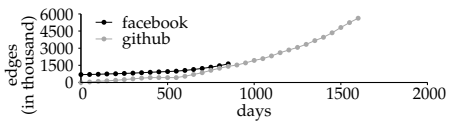


Figure 15: Evolution trends of a regional Facebook friendship graph and a GitHub collaboration graph

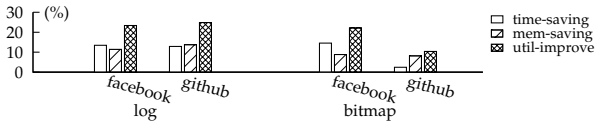


Figure 16: Performance of VT, *log*, and *bitmap* on Facebook and GitHub graphs

period then form the root versions of the two graphs. Figure 16 shows that VT outperforms both *log* and *bitmap* when executing SSSP over these two graphs, improving utility by 10.38–24.83%.

4.3 Discussion: Locality Revisited

At the core of VT lies the concept of locality. The effectiveness of VT depends on the high version access locality in multi-version workloads. Quantifying the locality of version access patterns, nevertheless, is difficult. In this paper, we express locality in terms of a range $n\delta$—defined by the number of deltas n and the size of delta δ—within which arbitrary version switching takes place. We have shown that VT achieves superior performance for a wide range of $n\delta$ configurations (cf. Figure 12), with respect to state of the art. Yet, VT's performance, as well as its relative gain with respect to other systems, needs careful investigation for other access patterns.

For example, for workloads featuring high computation-to-version-switching ratio and forward-only switching, we expect either *log* or a single-version system to perform the best. For such workloads, the significance of the computation-stage performance outweighs that of version switching. For example, when a large set of algorithms are applied to a loaded version, any version switching except the first one becomes a self-switching operation, incurring almost zero cost thanks to the memory management of supporting operating systems. In addition, forward-switching nullifies the need to preserve the graph representation of a version after it is processed. Thus a system optimized purely for high computation performance is favorable.[13] To efficiently handle such workloads, VT needs to be extended to support direct modification to the CSR, bypassing the shadowing effect of the delta cache. More importantly, the switching between existing operating modes of VT and this new direct modification mode, as well as other modes potentially devised in future work, requires a thorough investigation of the switching policies.

5 Conclusions

In this paper, we conducted a systematic investigation of the caching design space in multi-version graph processing scenarios, decomposing it into three dimensions: neighbor access efficiency, extensibility, and compactness. Our solution, Version Traveler, balances requirements from all three dimensions, achieving fast and memory-efficient version switching. It significantly outperforms PowerGraph and is superior to four multi-version reference designs.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Indranil Gupta, for their feedback.

[13] The relative merit of *log* to a single-version system is determined, in this case, by whether constructing the next version by modifying the current one is less costly than building it from scratch.

References

[1] Apache Giraph. `http://giraph.apache.org`. Retrieved in Apr. 2016.

[2] Github api. `https://developer.github.com/v3/`. Retrieved in Apr. 2016.

[3] Github archive. `https://www.githubarchive.org/`. Retrieved in Apr. 2016.

[4] AGMON BEN-YEHUDA, O., BEN-YEHUDA, M., SCHUSTER, A., AND TSAFRIR, D. The resource-as-a-service (raas) cloud. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Ccomputing* (Berkeley, CA, USA, 2012), HotCloud'12, USENIX Association.

[5] BHARDWAJ, A., BHATTACHERJEE, S., CHAVAN, A., DESHPANDE, A., ELMORE, A. J., MADDEN, S., AND PARAMESWARAN, A. G. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *7th Biennial Conference on Innovative Data Systems Research* (2015), CIDR '15.

[6] BHATTACHERJEE, S., CHAVAN, A., HUANG, S., DESHPANDE, A., AND PARAMESWARAN, A. Principles of dataset versioning: Exploring the recreation/storage tradeoff. *Proc. VLDB Endow. 8*, 12 (Aug. 2015), 1346–1357.

[7] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web* (2011), ACM Press.

[8] BOLDI, P., AND VIGNA, S. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)* (Manhattan, USA, 2004), ACM Press, pp. 595–601.

[9] BU, Y., BORKAR, V., JIA, J., CAREY, M. J., AND CONDIE, T. Pregelix: Big(ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment 8*, 2 (2014), 161–172.

[10] CHENG, R., HONG, J., KYROLA, A., MIAO, Y., WENG, X., WU, M., YANG, F., ZHOU, L., ZHAO, F., AND CHEN, E. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 85–98.

[11] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 17–30.

[12] GONZALEZ, J. E., XIN, R. S., DAVE, A., CRANKSHAW, D., FRANKLIN, M. J., AND STOICA, I. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), OSDI '14, USENIX Association, pp. 599–613.

[13] HAN, W., MIAO, Y., LI, K., WU, M., YANG, F., ZHOU, L., PRABHAKARAN, V., CHEN, W., AND CHEN, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2014), EuroSys '14.

[14] HOQUE, I., AND GUPTA, I. Lfgraph: Simple and fast distributed graph analytics. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems* (New York, NY, USA, 2013), TRIOS '13, ACM, pp. 9:1–9:17.

[15] KANG, U., TSOURAKAKIS, C. E., AND FALOUTSOS, C. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining* (Washington, DC, USA, 2009), ICDM '09, IEEE Computer Society, pp. 229–238.

[16] KHAYYAT, Z., AWARA, K., ALONAZI, A., JAMJOOM, H., WILLIAMS, D., AND KALNIS, P. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 169–182.

[17] KHURANA, U., AND DESHPANDE, A. Efficient snapshot retrieval over historical graph data. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Washington, DC, USA, 2013), ICDE '13, IEEE Computer Society, pp. 997–1008.

[18] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 591–600.

[19] KYROLA, A., BLELLOCH, G., AND GUESTRIN, C. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 31–46.

[20] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. Graphlab: A new parallel framework for machine learning. In *UAI* (2010), UAI '10, pp. 340–349.

[21] LU, Y., CHENG, J., YAN, D., AND WU, H. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment 8*, 3 (2014).

[22] MACKO, P., MARATHE, V., MARGO, D., AND SELTZER, M. LLAMA: Efficient graph analytics using large multiversioned arrays. In *Proceedings of the 2015 IEEE International Conference on Data Engineering (ICDE 2015)* (Washington, DC, USA, April 2015), ICDE '15, IEEE Computer Society.

[23] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[24] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 456–471.

[25] PEREZ, Y., SOSIČ, R., BANERJEE, A., PUTTAGUNTA, R., RAISON, M., SHAH, P., AND LESKOVEC, J. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1105–1110.

[26] REN, C., LO, E., KAO, B., ZHU, X., AND CHENG, R. On querying historical evolving graph sequences. *Proceedings of the VLDB Endowment 4*, 11 (2011), 726–737.

[27] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 472–488.

[28] SHEN, Y., CHEN, G., JAGADISH, H., LU, W., OOI, B. C., AND TUDOR, B. M. Fast failure recovery in distributed graph processing systems. *Proceedings of the VLDB Endowment 8*, 4 (2014).

[29] SHUN, J., AND BLELLOCH, G. E. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2013), PPoPP '13, ACM, pp. 135–146.

[30] TIAN, Y., BALMIN, A., CORSTEN, S. A., TATIKONDA, S., AND MCPHERSON, J. From "think like a vertex" to "think like a graph". *Proceedings of the VLDB Endowment 7*, 3 (2013).

[31] VISWANATH, B., MISLOVE, A., CHA, M., AND GUMMADI, K. P. On the evolution of user interaction in facebook. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)* (August 2009).

[32] ZHOU, C., GAO, J., SUN, B., AND YU, J. X. Mocgraph: Scalable distributed graph processing using message online computing. *Proceedings of the VLDB Endowment 8*, 4 (2014).

Tucana: Design and implementation of a fast and efficient scale-up key-value store

Anastasios Papagiannis[1], Giorgos Saloustros, Pilar González-Férez[2], and Angelos Bilas[1]
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
Email: {apapag,gesalous,pilar,bilas}@ics.forth.gr

Abstract

Given current technology trends towards fast storage devices and the need for increasing data processing density, it is important to examine key-value store designs that reduce CPU overhead. However, current key-value stores are still designed mostly for hard disk drives (HDDs) that exhibit a large difference between sequential and random access performance, and they incur high CPU overheads.

In this paper we present *Tucana*, a feature-rich key-value store that achieves low CPU overhead. Our design starts from a B^{ε}–tree approach to maintain asymptotic properties for inserts and uses three techniques to reduce overheads: copy-on-write, private allocation, and direct device management. In our design we favor choices that reduce overheads compared to sequential device accesses and large I/Os.

We evaluate our approach against RocksDB, a state-of-the-art key-value store, and show that our approach improves CPU efficiency by up to 9.2× and an average of 6× across all workloads we examine. In addition, *Tucana* improves throughput compared to RocksDB by up to 7×. Then, we use *Tucana* to replace the storage engine of HBase and compare it to native HBase and Cassandra two of the most popular NoSQL stores. Our results show that *Tucana* outperforms HBase by up to 8× in CPU efficiency and by up to 10× in throughput. *Tucana*'s improvements are even higher when compared to Cassandra.

1 Introduction

Recently, NoSQL stores have emerged as an important building block in data analytics stacks and data access in general. Their main use is to perform lookups based on a key, typically over large amounts of persistent data and over large numbers of nodes. Today, Amazon uses Dynamo [16], Google uses BigTable [9], Facebook and Twitter use both Cassandra [29] and HBase [2].

The core of a NoSQL store is a key-value store that performs (key,value) pair lookup. Traditionally key-value stores have been designed for optimizing accesses to hard disk drives (HDDs) and with the assumption that the CPU is the fastest component of the system (compared to storage and network devices). For this reason, key-value stores tend to exhibit high CPU overheads. For instance, our results show that popular NoSQL stores, such as HBase and Cassandra, require several tens or hundreds of thousands of cycles per operation. For relatively small data items we therefore need several modern cores to saturate a single 1 Gbit/s link or equivalently a 100-MB/s-capable HDD. Given today's limitations in power and energy and the need to increase processing density, it is important to examine designs that not only exhibit good device behavior, but also improve host CPU overheads.

Our goal in this paper is to draw a different balance between device and CPU efficiency. We start from a B^{ε}–tree [7] approach to maintain the desired asymptotic properties for inserts, which is important for write-intensive workloads. B^{ε}–trees achieve this amortization by buffering writes at each level of the tree. In our case, we assume that the largest part of the tree (but not the data items) fit in memory and we only perform buffering and batching at the lowest part of the tree. Then, we develop a design that manages variable size keys and values, deals with persistence, and stores data directly on raw devices.

Although we still use the buffering technique of B^{ε}–trees to amortize I/Os, we take a different stance with respect to randomness of I/Os. Unlike LSM-trees [43], we do not make an effort to generate large I/Os. LSM-trees produce large I/Os by maintaining large sorted containers of data items in memory, which can then be read or writ-

[1]Also with the Department of Computer Science, University of Crete, Greece.

[2]Also with the Department of Computer Engineering, University of Murcia, Spain.

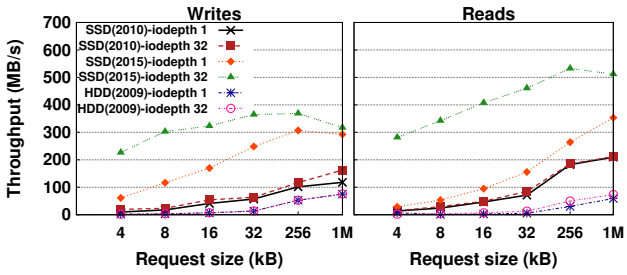


Figure 1: Throughput vs. block size for one HDD and two SSDs, measured with FIO [3].

ten as a whole. These large sorted containers are maintained via a compaction technique that relies on sorting and merging smaller pieces. Although this approach has proven extremely effective for HDDs, it results in high CPU overheads and I/O amplification, as we show in our evaluation for LSM-trees.

New storage technologies, such as flash-based solid-state drives (SSDs) and non-volatile memory (NVM) are already part of the I/O hierarchy with increasing use. Such devices decrease the role of randomness in data accesses. Figure 1 shows throughput for random I/Os on two different generations of SSDs and a HDD for different queue depths and request sizes. We see that HDDs achieve peak throughput for request sizes that approach 1 MB for both reads and writes. Increasing the number of outstanding I/Os does not provide a significant benefit. On the other hand, a commodity SSD of 2015 achieves both maximum write throughput and more than 90% of the maximum read throughput at 32 outstanding requests of 32 KB size. The 2010 SSD has roughly the same behavior with 256 KB requests. Allowing a higher degree of randomness enables us to reduce read and write traffic amplification in the design of the key-value store, which has significant cost in terms of CPU and memory.

We design a full featured key-value store, *Tucana*, that achieves lower host CPU overhead per operation than other state-of-the-art systems. *Tucana* provides persistence and recovery from failures, arbitrary dataset sizes, variable key and value sizes, concurrency, multi-threading, and versioning. We use copy-on-write (CoW) to achieve recovery without the use of a log, we directly map the storage device to memory to reduce space (memory and device) allocation overhead, and we organize internal and leaf nodes similar to traditional approaches [11] to reduce CPU overhead for lookup operations.

To evaluate our approach, we first compare with RocksDB, a state-of-the-art key-value store. Our results show that *Tucana* is up to 9.2× better in terms of cycles/op and between $1.1\times$ *to* $7\times$ in terms of ops/s, across all workloads. This validates our hypothesis that randomness is less important for SSD devices, when there is an adequate degree of concurrency and relatively small I/O requests.

To examine the impact of our approach in the context of real systems, we use *Tucana* to improve the throughput and efficiency of HBase [2], a popular scale-out NoSQL store. We replace the LSM-based storage engine of HBase with *Tucana*. Data lookup, insert, delete, scan, and key-range split and merge operations are served from *Tucana*, while maintaining the HBase mapping of tables to key-value pairs, client API, client-server protocol, and management operations (failure handling and load balancing). The resulting system, H-*Tucana*, remains compatible with other components of the Hadoop ecosystem. We compare H-*Tucana* to HBase and Cassandra using YCSB and we find that, compared to HBase, H-*Tucana* achieves between $2-8\times$ better CPU cycles/op and $2-10\times$ higher operation rates across all workloads. Compared to Cassandra, H-*Tucana* achieves even higher improvements.

Our specific contributions in this work are:

- The design and implementation of a key-value data store that draws a different balance between device behavior and host overheads.
- Practical B^{ε}–tree extensions that leverage mmap-based allocation, copy-on-write, and append-only logs to reduce allocation overheads.
- An evaluation of existing, state-of-the-art, persistent key-value stores and a comparison with *Tucana*, as well as an improved implementation of HBase.

The rest of this paper is organized as follows: Section 2 provides an overview of persistent data structures. Section 3 describes our design. Section 4 presents our evaluation methodology and our experimental analysis. Section 5 reviews prior related work. Finally, Section 6 concludes the paper.

2 Background

Persistent stores can be categorized into four groups: key-value stores [18, 22, 25], NoSQL stores [2, 29], document DBs [12], and graph DBs [37, 52]. The last two categories are generally more domain specific. In this work we target the first two and we use the term *stores* to refer collectively to both categories.

The abstraction offered by key-value stores is typically a flat, object-like abstraction, whereas NoSQL stores offer a table-based abstraction, closer to relational concepts. The operations supported by such stores, regardless of the abstraction used, consist of simple *dictionary*

operations: get(), put(), scan(), and delete(), with possible extensions for versioned items and management operations, key-range split and merge. Although this is a simple abstraction over stored data, it has proven to be powerful and convenient for building modern services, especially in the area of data processing and analytics.

State-of-the-art stores [2, 18, 22, 29] have been designed primarily for HDDs and typically use at their core an LSM-tree structure. LSM-trees [43] are a write-optimized structure that is a good fit for HDDs where there is a large difference in performance between random and sequential accesses. LSM-trees organize data in multiple levels of large, sorted containers, where each level increases the size of the container. Additionally, small amounts of search metadata, such as Bloom filters, are used for accelerating scan and get operations.

This organization has two advantages. First, it requires little search metadata because containers are sorted and therefore, practically all I/Os generated are related to data items (keys and values). Second, due to the container size, I/Os can be large, up to several MB each, resulting in optimal HDD performance. The drawback is that for keeping large sorted containers they perform compactions which (a) incurs high CPU overhead and (b) results in I/O amplification for reads and writes.

Going forward device performance and CPU-power trends dictate different designs. In this work, we use as a basis a variant of B-trees, broadly called B$^\varepsilon$–trees [7].

B$^\varepsilon$–trees are B-trees with an additional per-node buffer. By using these buffers, they are able to batch insert operations to amortize their cost. In B$^\varepsilon$–trees the total size of each node is B and ε is a design-time constant between [0,1]. ε is the ratio of B that is used for buffering, whereas the rest of the space in each node (1-ε) is used for storing pivots.

Buffers contain messages that describe operations that modify the index (insert, update, delete). Each such operation is initially added to the tree's root node buffer. When the root node buffer becomes full, the structure uses the root pivots to propagate a subset of the buffered operations to the buffers of the appropriate nodes at the next level. This procedure is repeated until operations reach a leaf node, where the key-value pair is simply added to the leaf. Leaf nodes are similar to B-Trees and they do not contain an additional buffer, beyond the space required to store the key-value pairs. The cost of an insertion in terms of I/Os is $O(\frac{\log_B N}{\varepsilon B^{1-\varepsilon}})$, where a regular B-Tree has $O(\log_B N)$ [7, 26].

A get operation is similar to a B-Tree. It traverses the path from the root to the corresponding leaf. This results in similar complexity to B-trees, regarding I/O operations. The main difference is that in a B$^\varepsilon$–tree we also need to search the buffers of the internal nodes along the path. A range scan is similar to a get, except that messages for the entire range of keys must be checked and applied as the appropriate subtree is traversed. Therefore, buffers are frequently modified and searched. For this reason, they are typically implemented with tree indexes rather than sorted containers.

Compared to LSM-trees, B$^\varepsilon$–trees incur less I/O amplification. B$^\varepsilon$–trees use an index, compared to LSM-trees, in order to remove the need for sorted containers. This results in smaller and more random I/Os. As device technology reduces the I/O size required to achieve high throughput, using a B$^\varepsilon$–tree instead of an LSM-tree is a reasonable decision.

Next, we present the design of *Tucana*, a key-value store that aims to significantly improve the efficiency of data access.

3 *Tucana* Design

Figure 2 shows an overview of *Tucana*. More specifically, Figure 2a shows the index organization, which uses B$^\varepsilon$–trees as a starting point (Section 3.1). In Figure 2b we depict *Tucana*'s approach for allocation and persistence, which we discuss in Sections 3.2 and 3.3, respectively.

3.1 Tree Index

Figure 3 shows the differences between *Tucana* and a B$^\varepsilon$–tree. On the left side of the figure we show a B$^\varepsilon$–tree, which we explain in Section 2. On the right side of the figure we show *Tucana*, where we distinguish nodes that fit in main memory from those that do not. To improve host-level efficiency (in terms of cycles/op), *Tucana* limits buffering and batching to the lowest part of the tree. In many cases today, the largest part of the index structure (but not the actual data) fits in main memory (DRAM today and byte-addressable NVM in the future) and therefore, we do not buffer inserts in intermediate nodes. *Tucana* design provides desirable asymptotic properties for random inserts, where a single I/O is amortized over multiple insert operations. On the other hand, B$^\varepsilon$–trees generate smaller I/Os with higher randomness compared to LSM-trees. However, they do not require compaction operations and incur lower I/O amplification. Using fast storage devices we can trade compactions with smaller random I/Os, compared to what an LSM-tree produces, without affecting device performance.

Figure 2a shows the index organization in *Tucana*. The index consists of internal nodes with pointers to next level nodes and pointers to variable size keys (pivots). We use a separate space per internal node to store the variable size keys themselves. Pointers to keys are sorted based on the key, whereas keys are appended to

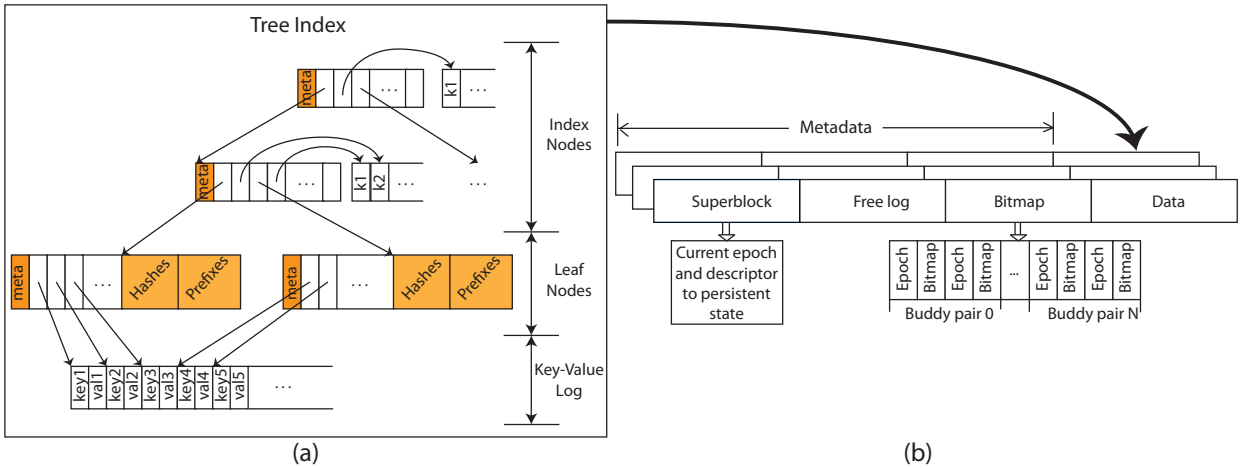


Figure 2: The top-level design of *Tucana*. The left part (a) of the figure shows the tree index. The right part (b) shows the volume layout.

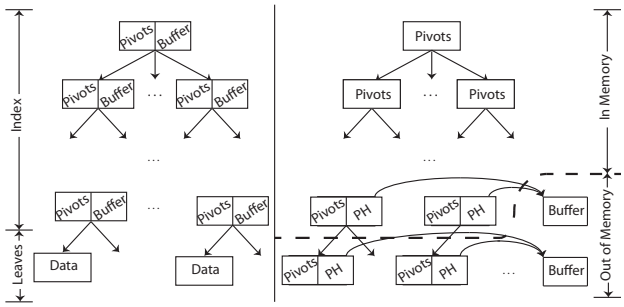


Figure 3: Comparison of B$^\epsilon$–tree (left) and *Tucana* (right). In *Tucana* we distinguish the part of the tree that fits in memory above the dashed line and the rest that does not. PH stands for Prefix-Hash.

the buffer. The leaf nodes contain sorted pointers to the key-value pairs. We use a single append-only log to store both the key and values. The design of the log is similar to the internal buffers of B$^\epsilon$–trees.

Insert operations traverse the index in a top down fashion. At each index node, we perform a binary search over the pivots to find the next level node to visit. When we reach the leaf, we append the key-value pair to the log and we insert the pointer in the leaf, keeping pointers sorted by the corresponding key. Then, we complete the operation. Compared to B$^\epsilon$–trees we avoid the buffering at intermediate nodes. If a leaf is full, we trigger a split operation prior to insert. Split operations, in index or leaf nodes, produce two new nodes each containing half of the keys and they update the index in a bottom-up fashion. Delete operations place a tombstone for the respective keys, which are removed later. Deletes will eventually cause rebalancing and merging [4].

Point queries traverse the index similar to inserts to locate the appropriate leaf. At the leaf, we perform a binary search to locate the pointer to the key-value pair. Since there are no intermediate buffers as in B$^\epsilon$–trees, we do not need to perform searches in the intermediate levels. Finally, range queries locate the starting key similar to point queries and subsequently use the index to iterate over the key range. It is important to notice that in contrast to B$^\epsilon$–trees we do not need to flush all the intermediate buffers prior to a scan operation.

We note that binary search in the leaf nodes and index nodes is a dominant function used by all operations. To reduce memory footprint for metadata, *Tucana* does not store keys in leaves. This means that keys during binary search need to be retrieved from the device. To avoid this, *Tucana* uses two optimizations, prefixes and hashes.

We store as metadata, a fixed-size prefix for each key in the leaf block. Binary search is performed using these prefixes, except when they result in ambiguity, in which case the entire key is fetched from the log. Prefixes improve performance of inserts, point queries, and range queries. In our tuning of prefixes we find that for different types of keys, prefixes eliminate 65%–75% of I/Os during binary search in leaves.

Additionally, a hash value for each key is stored in the leaf nodes. Hashes help with point queries. For a point query we first do a binary search over prefixes. If this results in a tie, then we linearly examine the corresponding (so not all) hashes. We use Jenkins hash function (one-at-a-time) [27] to produce 4-byte hashes. Then the key is read to ensure there is no collision. In our experiments we find that hashes identify the correct key-value pair in more that the 98% of the cases.

The memory footprint can be analyzed as follows. As-

sume N is the number of keys in the dataset, C is the number of pointers in internal nodes and L in leaves, B is the block size used for internal nodes and leaves, and $h = \log_C N/L$ is the height of the tree. S_k and S_v are the average sizes for keys and values respectively. The size (bytes) of the different components of the index are:

$$dataset = (S_k + S_v)N, \tag{1}$$

$$\begin{aligned} full\ index &= internal\ nodes + leaves \\ &= (B + CS_k)\sum_{i=0}^{h-1} C^i + B\frac{N}{L}, \end{aligned} \tag{2}$$

$$\frac{index}{dataset} = \frac{L(B + CS_k) + BC}{CL(S_k + S_v)}. \tag{3}$$

Equation 3 shows the ratio of the index to the dataset. Now, if we consider that C and L are similar (e.g. in our implementation we use C=230 and L=192) and that B/C is the pointer size, which typically is 8 bytes, we have:

$$\frac{index}{dataset} = \frac{16 + S_k}{S_k + S_v} \tag{4}$$

If we consider that the size of values (S_v) is at least 20× larger than the size of the keys (S_k) and that key size is at least 16 bytes, the index size is around 10% the size of the dataset. Given current cost per GB for DRAM and FLASH, and if we assume that a server spends roughly the same cost for DRAM and FLASH, it is reasonable to assume that the index fits in memory, especially in servers used for data analytics.

For cases when the index fits in memory and the dataset does not, then each search requires one I/O. For inserts, I/O operations for consecutive random inserts are amortized due to the append log.

In case where the index starts to exceed memory, more I/Os are required for each search and insert. Since internal nodes and leaves store pointers to keys and keys are stored in a log, we need two I/Os to read or update an arbitrary item at the bottom of the tree. In this case we need to introduce buffering at additional levels of the index.

3.2 Device layout and access

Figure 2b depicts the data layout in *Tucana*. *Tucana* manages a set of contiguous segments of space to store data. Each segment can be a range of blocks on a physical, logical, virtual block device, or a file. To reduce overhead, segments should be allocated directly on virtual block devices, without the use of a file system. Our measurements show that using XFS as the file system results in a 5-10% reduction in throughput compared to using a virtual block device directly without any file system.

Each segment is composed of a metadata portion and a data portion. The metadata portion contains the superblock, the free log, and the segment allocator metadata (bitmap). The superblock contains a reference to a descriptor of the latest persistent and consistent state for a segment. Modifying the superblock commits the new state for the segment. Each segment has a single allocator common for all databases (key ranges) in a segment. The data portion contains multiple databases. Each database is contained within a single segment and uses its own separate indexing structure.

The allocator keeps persistent state about allocated blocks of a configurable size, typically set to 4 KB, and multiples of it. For this purpose, it uses bitmaps because in key-value stores allocations can be in the order of KBs, as opposed to filesystems that typically do larger allocations. Moreover, allocator bitmaps are accessed directly via an offset and at low overhead, while for searches there are efficient bit parallel techniques [8]. It also maintains state about free operations and performs them lazily in a log structure named *Free log*.

In all persistent key-value stores, including *Tucana*, the index includes pointers to data items in the storage address space. During system operation, part of the index and data are cached in memory. When traversing the index to serve an operation, there is a need to translate storage pointers to pointers in memory. This leads to frequent cache lookups that cannot be avoided easily. Essentially, the cache serves as a mechanism to translate pointers from the storage to the memory address space. Previous work [24] indicates that when all data and metadata fit in memory, managing this cache requires about one-third of the index CPU cycles.

Most key-value stores today follow this caching approach [2, 18, 22, 29, 41]. This allows the key-value store to also control the size and timing of I/O operations between the memory cache and the storage devices, as well as the cache policy.

Instead, *Tucana* uses an alternative approach based on `mmap`. `mmap` uses a single address space for both memory and storage and virtual memory protection to determine the location (memory or storage) of an item. This eliminates the need for pointer translation at the expense of page faults. We note that pointer translation occurs during index operations regardless of whether items are in memory or not, whereas page faults occur only when items are not in memory. The use of `mmap` also allows *Tucana* to use a single allocator for memory and device space management. Additionally, `mmap` eliminates data copies between kernel and user space.

The use of `mmap` has three drawbacks. First, each write operation of variable size is converted to a read-modify-write operation, increasing the amount of I/O. In our design, due to the copy-on-write persistence (see Sec-

tion 3.3), all writes modify eventually the full page and there can be no reads to unwritten parts of a page. Therefore, we use a simple filter block device in the kernel, which filters read-before-write operations and merely returns a page of zeros. Write and read-after-write operations are not filtered and are forwarded to the actual device. The filter module uses a simple, in-memory bitmap and is initialized and updated by *Tucana* via a set of ioctls. The size of the in-memory bitmap is proportional to the block device size (for 1 TB of storage we need 32 MB of memory).

Second, `mmap` results in the loss of control over the size and timing of I/O operations. `mmap` generates page-sized I/Os (4 KB). To mitigate the impact of small I/Os we use `madvise` to instruct `mmap` to generate larger I/Os. To control their timing we use `msync` for specific items and memory ranges during commit operation.

Third, `mmap` introduces page faults for fetching data. The number of page faults depends on `mmap` kernel page eviction policy. *Tucana* would benefit from custom eviction policies that keep the index and the tail of the append log in memory. In this work, we do not make an attempt to control these policies. However, future work should examine this issue in more detail.

3.3 Copy-on-write persistence

Tucana uses a Copy-on-Write (CoW) approach for persistence instead of a Write-Ahead-Log (WAL). WAL produces sequential write I/Os at the expense of doubling the amount of writes (in the log and later in place). CoW performs only the necessary writes, however, it generates a more random I/O pattern. Therefore, although a WAL is more appropriate for HDDs, CoW has more potential for fast devices. The use of CoW is also motivated by three additional reasons; (a) It is amenable to supporting versioning. (b) It allows instantaneous recovery, without the need to redo or undo a log. (c) It helps increase concurrency by avoiding lock synchronization for different versions of each data item [33], as we discuss in the next subsection.

The state of a segment consists of the allocator, tree metadata, and buffers. CoW is used to maintain the consistency of both allocator and tree metadata. The bitmap in each segment is organized in *buddy pairs*, as shown in Figure 2b. Each *buddy pair* consists of two 4 KB blocks that contain information about allocated space. Each buddy is marked with a global per segment increasing counter named ***epoch***. The epoch field is incremented after a successful commit operation and denotes the latest epoch in which the buddy was modified. At any given point only one buddy of the pair is active for write operations, whereas the other buddy is immutable for recovery. Commits persist and update modified buddy pairs.

The allocator defers free operations with the use of the free log [6]. Directly applying a free operation that could be rolled back in the presence of failures is more complicated as it can corrupt persistent state. We log free operations using their epoch id, and we perform them later after their epoch becomes persistent.

To maintain the consistency of the tree structure during updates, each internal index and leaf node uses epochs to distinguish its latest persistent state. During an update, the node's epoch indicates whether a node is immutable, in which case a CoW operation takes place. After a CoW operation for inserting a key, the parent of the node is updated with the new node location in a bottom-up fashion. The resulting node belongs to epoch+1 and will be persisted during the next commit. Subsequent updates to the same node before the next commit are batched by applying them in place. Since we store keys and values in buffers in an append-only fashion, we need to only perform CoW on the header of each internal node.

Tucana's persistence relies on the atomic transition between consistent states for each segment. Metadata and data in *Tucana* are written asynchronously to the devices. However, transitions from state to state occur atomically via synchronous updates to the segment's superblock with `msync` (commits). Each commit creates a new persistent state for the segment, identified by a unique epoch id. The epoch of the latest persistent state of a segment is stored in a descriptor to which the superblock keeps a reference.

Commits can take place in parallel with read and write operations. To achieve this, a commit is performed in two steps: (1) Initially, it marks the current state as persistent by increasing the epoch of the system. This state includes the bitmap and the tree indexes for this segment. (2) It flushes the state of the segment to the device. In case of a failure during a commit, the segment simply rolls back to the latest persistent state by ignoring any writes that have reached the device but were not committed via the metadata epoch states.

During a commit operation, the bitmap cannot be modified by new allocations (a subset of the write operations) because this may change the state on the device (`mmap` may propagate any write from memory to the device asynchronously). In case the current commit fails, then both *buddy pairs* will be inconsistent. To avoid this, allocations during a commit are buffered in a temporary location in memory and are applied at the end of the commit.

3.4 Concurrency in *Tucana*

Concurrency in key-value stores is important for scaling up as server density increases in terms of CPU, storage,

and network throughput. Key-value stores typically operate under high degrees of concurrency, due to the large numbers of client requests.

Similar to most key-value stores, *Tucana* partitions datasets in multiple databases (key ranges). Requests in different ranges can be served without any synchronization. The only exception in *Tucana* is insert operations in different regions that are stored in the same segment. In this case the existence of a single segment allocator requires synchronization across ranges during allocation operations. To reduce the impact of such synchronization, the allocator operates in a batched mode, where a request reserves more space than required for the current operation. Subsequent inserts to the same database do not need to request space from the allocator.

Within each range, *Tucana* allows any number of concurrent reads and a single write without synchronization. To achieve this, *Tucana* uses the versions of the segment created through commits, similar to read-copy-update synchronization [38]. In particular, we serve read operations from the latest persistent version of the segment, which is immutable. Writes on the other hand are served from the modified root which contains all modifications.

Updates applied by an application are visible to readers after a commit. *Tucana*'s API offers additional fence operations to allow higher layers to control when updates become visible.

Finally, in the current state of the prototype, *Tucana* does not allow multiple concurrent writes in the same range. Although there are possible optimizations, especially to allow non-conflicting writes via copy-on-write, or dynamic partitioning of the key-space, we leave these for future work.

3.5 H-*Tucana*

HBase [2] is a scale-out columnar store which supports a small and volatile schema. HBase offers a table abstraction over the data, where each table keeps a set of key-value pairs. Each table is further decomposed into regions, where each region stores a contiguous segment of the key space. Each region is physically organized as a set of files per column, as shown in Figure 4.

At its core HBase uses an LSM-tree to store data [43]. We use *Tucana* to replace this storage engine, while maintaining the HBase metadata architecture, node fault tolerance, data distribution and load balancing mechanisms. The resulting system, H-*Tucana*, maps HBase regions to segments (Figure 4), while each column maps to a separate tree in the segment. In our work, and to eliminate the need for using HDFS under HBase, we modify HBase so that a new node handles a segment after a failure. We assume that segments are allocated over a reliable shared block device, such as a storage area network (SAN) or virtual SAN [39, 50] and are visible to all nodes in the system. In this model, the only function that HDFS offers is space allocation. *Tucana* is designed to manage space directly on top of raw devices, therefore, it does not require a file system. H-*Tucana* assumes the responsibility of elastic data indexing, while the shared storage system provides a reliable (replicated) block-based storage pool.

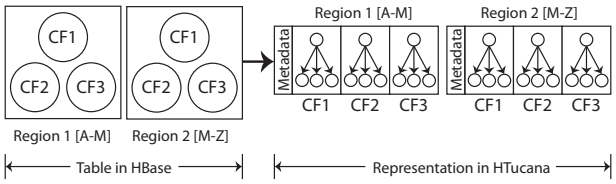


Figure 4: Table storage in HBase and H-*Tucana*. CF stands for column family.

4 Experimental evaluation

In this section, we compare *Tucana* to RocksDB [18] and H-*Tucana* to HBase [2] and Cassandra [29]. *Tucana* and RocksDB support similar features including persistence and recovery, arbitrary size keys and values and versions. In the same category there are other popular key-value stores, such as LevelDB, KyotoDB, BerkeleyDB, and PerconaFT (based on Fractal Index Trees). In our experiments we find that RocksDB outperforms all of them [19, 23] and therefore, we present only the comparison between *Tucana* and RocksDB.

HBase and Cassandra are NoSQL databases that are widely used as a back-end for high throughput systems. HBase and Cassandra use LSM-trees [43].

4.1 Methodology

Our experimental platform consists of two systems (client and server) each with two quad-core Intel(R) Xeon(R) E5520 CPUs running at 2.7 GHz. The server is equipped with 48 GB DDR-III DRAM, and the client with 12 GB. Both nodes are connected with a 10 Gbits/s network link. As storage devices, the server uses four Intel X25-E SSDs (32 GB) and we make a RAID-0 with them using the standard *md* Linux driver. *Tucana* is implemented in C and can be accessed from applications as a shared library. H-*Tucana* is cross-linked between the Java code of HBase and the C code of *Tucana*.

We use the open-source Yahoo Cloud Serving Benchmark (YCSB) [13] to generate synthetic workloads. The default YCSB implementation executes gets as range queries and therefore, exercises only scan operations.

Workload	
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

Table 1: Workloads evaluated with YCSB. All workloads use a query popularity that follows a Zipf distribution except for D that follows a latest distribution.

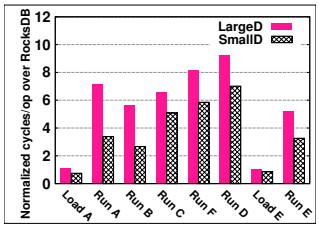
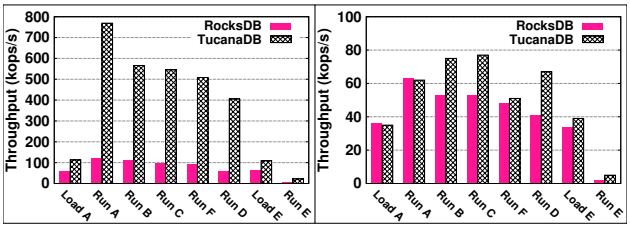


Figure 5: *Tucana* improvement compared to RocksDB in cycles per operation.



(a) Small dataset (b) Large dataset

Figure 6: Performance of *Tucana*DB and RocksDB in ops/s.

For this reason, we modify YCSB to use point queries for get operations. Range queries are still exercised in Workload E, which uses scan operations.

When comparing RocksDB and *Tucana* we use a low-overhead C++ version of YCSB-C [13, 45]. The original Java YCSB benchmark requires JNI to run with RocksDB and *Tucana*, which are written in C++ and C respectively, incurring high overheads.

In all cases, we run the standard workloads proposed by YCSB with the default values. Table 1 summarizes these workloads. We run the following sequence proposed by the YCSB author: Load the database using workload A's configuration file, run workloads A, B, C, F, and D in a row, delete the whole database, reload the database with workload E's configuration file, and run workload E.

When comparing *Tucana* to RocksDB we use 256 YCSB threads and 64 databases (unless noted otherwise) and we choose the appropriate database by hashing the keys. When comparing H-*Tucana* to HBase and Cassandra we use 128 YCSB threads and 8 regions for HBase and H-*Tucana*. Cassandra is hash-based and does not support the notion of region, so we use a single table.

We use a small dataset that fits in memory and a large dataset that does not. The small dataset is composed of 60M or 100M records when using *Tucana* and H-*Tucana*, respectively. The large dataset has 300M or 500M records when using *Tucana* and H-*Tucana*, respectively.

In all the cases, the load phase creates the whole dataset and the run phases issue 5 million operations, bounded also by time (one hour max). With *Tucana*, even in the case of the large dataset the index nodes fit in memory as per our assumptions.

We measure efficiency as cycles/op, which shows the cycles needed to complete an operation on average. We calculate efficiency as:

$$cycles/op = \frac{\frac{CPU\_utilization}{100} \times \frac{cycles}{s} \times cores}{\frac{average\_ops}{s}}, \quad (5)$$

where $CPU\_utilization$ is the global average of CPU utilization among all processors, excluding idle and I/O time, as given by `mpstat`. As cycles/s we use the per-core clock frequency. $average\_ops/s$ is the throughput reported by YCSB and *cores* is the number of cores including hyperthreads.

4.2 Efficiency of *Tucana*

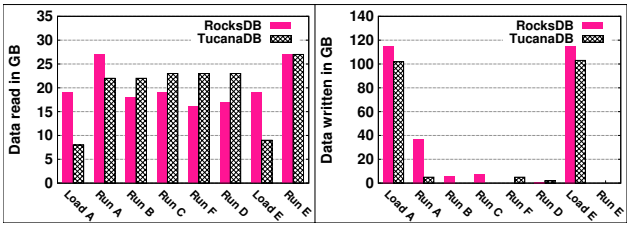
Figure 5 shows the improvement over RocksDB in efficiency. In workloads Load A and Load E that are insert intensive, *Tucana* is similar to RocksDB for both small and large datasets, since both use write-optimized data structures. In all other workloads *Tucana* outperforms RocksDB by 5.2× to 9.2× for the small dataset and by 2.6× to 7× for the large dataset.

We note that increased efficiency can also be achieved with low absolute performance, which is not desirable. Figure 6 shows ops/s for the two systems. We see that for the small dataset *Tucana* outperforms RocksDB by 2× to 7× and by 4.47× on average in absolute performance (throughput) as well. For the large dataset, where both systems are limited by device performance, *Tucana* outperforms RocksDB by 1.1× to 2.1× and by 1.35× on average. Average SSD utilization for all workloads is 93% for *Tucana* and 78% for RocksDB. *Tucana* has on average smaller request size, 86 KB compared to 415 KB for RocksDB. As next generation SSDs close the gap be-

Inserts	Write (GB)	rq_sz	SSD (2010) time (s)	SSD (2015) time (s)
Tucana	123	18K	133	31
RocksDB	435	884K	623	100
Speedup			4.68	3.22

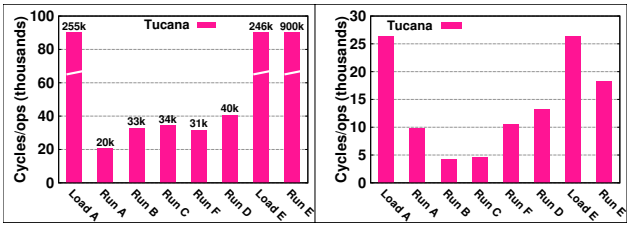
Inserts	Read (GB)	rq_sz	SSD (2010) time (s)	SSD (2015) time (s)
Tucana	26	4K	256	140
RocksDB	29	6K	229	171
Speedup			0.89	1.22

Table 2: Performance for the traffic pattern induced by *Tucana* and RocksDB as modeled with FIO to isolate device behavior.



(a) Reads (b) Writes

Figure 7: Total amount of data read and written during each YCSB workload.



(a) 10 qualifiers (b) 1 qualifier

Figure 8: Number of cycles needed for YCSB workloads.

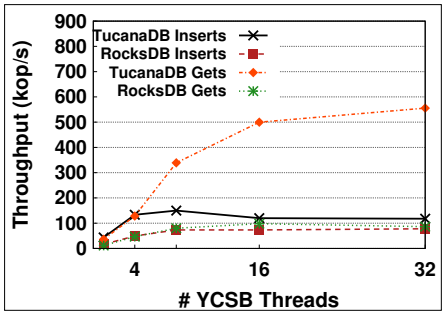


Figure 9: Scalability of RocksDB and *Tucana*DB with increasing threads, using the small dataset.

tween sequential and random performance, we expect even larger performance improvements over RocksDB and similar stores.

Next, we examine I/O amplification and randomness. We run an insert-only benchmark (random distribution) using a *single* database of size 36.3 GB. RocksDB writes 435 GB while *Tucana* writes 123 GB, thus 3.5× less than RocksDB. Due to compaction operations, RocksDB also reads 2.3× the amount of data read by *Tucana*, 69 GB vs. 29 GB. Table 2 shows the performance difference between these two patterns on two different SSD generations, using FIO (Flexible I/O) [3] to generate each pattern. For inserts, *Tucana*'s I/O pattern is 4.68× faster on the older SSD (2010) and 3.22× faster on the newer SSD (2015), compared to RocksDB's I/O pattern and volume. For gets, the difference in volume size and request size is lower and performance differences are smaller. The I/O pattern of RocksDB is better by 11% for the older SSD, whereas the I/O pattern of *Tucana* is better by 22% for the newer SSD.

Figure 7 shows read and write amplification using 64 databases. Although *Tucana* incurs less I/O on average for both read and write, the difference with RocksDB in this case is smaller. On average RocksDB writes 3.33× and reads 1.25× more data.

Next, we examine the absolute number of cycles/op for each workload (Figure 8a). Each operation is a composite operation over a row with ten qualifiers and therefore a get operation performs ten lookup operations. For this reason, we also present numbers for the same workloads, with one qualifier per row in Figure 8b. In addition, in the case of Workload E the default average length of a range query is fifty. In Figure 8b we change the scan length to five. On average, an insert operation takes about 26K cycles (Load A & Load E), a point query (get) 4K cycles (Run C) and a range query (scan), including initialization and five rows of one qualifier about 18K cycles (Run E). The other workloads are mixes of these operations. If we examine a breakdown of cycles, we see that on average 15% is used by YCSB, 43% by *Tucana*, 38% by the OS kernel, and 4% by other server processes. More specifically, for an insert-only benchmark 35% is used by *Tucana* and 60% by OS kernel. On the other hand for a get-only benchmark 66% is used by *Tucana* and 26% by kernel. System time is due to `mmap` that handles page faults, mappings, and the swapper that evicts dirty pages to devices.

Finally, Figure 9 shows scalability of *Tucana* and RocksDB with the number of server cores. We use the small dataset that fits in memory, partitioned in 64

databases, and we increase the load by increasing the number of YCSB threads that issue requests. For gets, *Tucana* is able to scale and it saturates the full server at 16 YCSB threads. *Tucana* provides lock-less gets and therefore uses all available cores. After warm-up, where data is brought in memory, system utilization is about 100% at 16 YCSB threads. On the other hand, RocksDB, even after warm up, still has about 25% idle CPU time at 8 or more YCSB threads, indicating synchronization bottlenecks.

For inserts, *Tucana* saturates the server at about 8 YCSB threads, where CPU is utilized at 90-95%. RocksDB scales up to 8 threads also, where it saturates the server. Due to its more random I/O pattern, *Tucana* incurs higher device utilization, about 50% vs. 20% for RocksDB. Generally, scaling for puts in both systems is related to the number of databases. In this work, we do not explore this dimension further.

4.3 Impact on NoSQL store performance

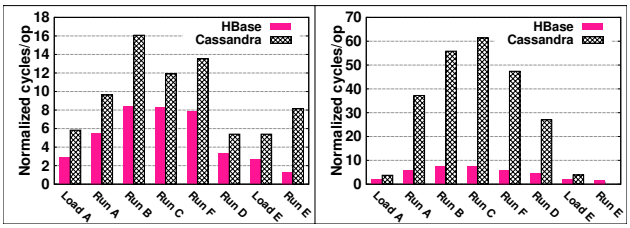
In this section, we analyze the efficiency and performance of H-*Tucana*, compared to HBase [2] and Cassandra [29].

Figure 10 depicts the speedup in efficiency (cycles/op) achieved by H-*Tucana* over HBase and Cassandra. We see that H-*Tucana* significantly outperforms both HBase and Cassandra. Compared to HBase, H-*Tucana* uses fewer cycles/op by up to 2.9×, 8.4×, and 5.6× for write-intensive, read intensive, and mixed workloads. Compared to Cassandra, the improvement depends on the size of the dataset. With the small dataset H-*Tucana* outperforms Cassandra by up to 5.8×, 16.1×, and 13.5× for the write, read intensive, and mix workloads, respectively. With the large dataset, H-*Tucana* improves cycles/op over Cassandra by up 3.9×, 61.4×, and 37.2× write, read-intensive and mixed workloads respectively.

Next, we examine throughput in terms of ops/s. Figure 11 shows performance in kops/s whereas Figure 12 depicts the amount of data read and written by each workload.

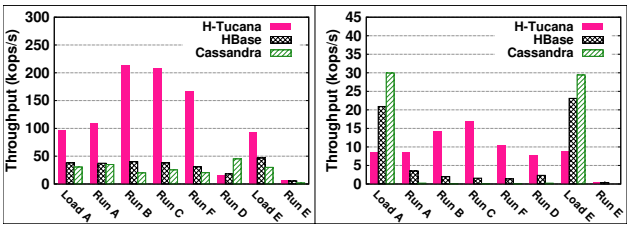
For the small dataset, H-*Tucana* has up to 5.4× higher throughput compared to HBase, and up to 10.7× compared to Cassandra. In addition, H-*Tucana* does not perform any reads during the run phases. Cassandra does not read any data either, whereas HBase reads 5.1 GB and 5.2 GB when running workloads A and E, respectively. The amount of data written to the device is significantly reduced by H-*Tucana* by 38% and 17% compared to HBase and Cassandra.

For the large dataset, during the run phase, H-*Tucana* outperforms HBase and Cassandra by up to 10.7× and 153.3×, respectively. This improvement is reflected in a significant reduction of the amount of data read from



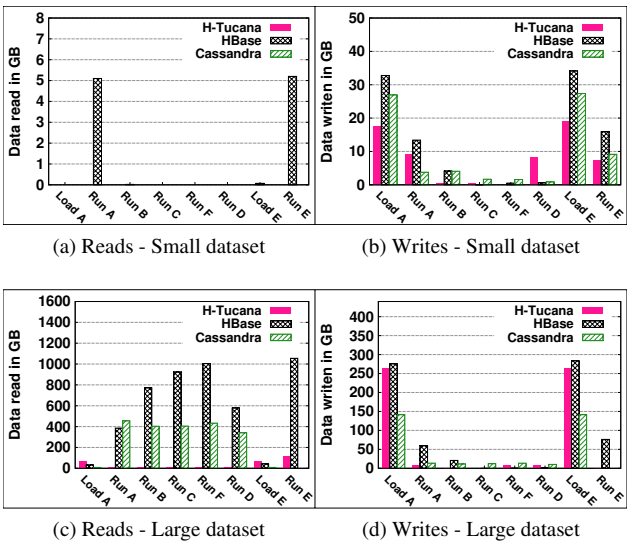
(a) Small dataset (b) Large dataset

Figure 10: Improvement in efficiency (cycles/op) achieved by H-*Tucana* over HBase and Cassandra.



(a) Small dataset (b) Large dataset

Figure 11: Throughput (kops/s) achieved by H-*Tucana*, HBase and Cassandra.



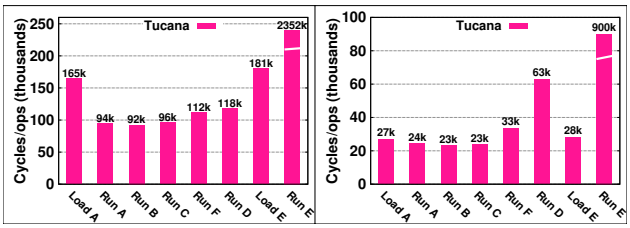
(a) Reads - Small dataset (b) Writes - Small dataset

(c) Reads - Large dataset (d) Writes - Large dataset

Figure 12: Amount of data, in GB, read/written by H-*Tucana*, HBase, and Cassandra.

the storage device, by up to 16× and 6.9× compared to HBase and Cassandra, respectively. For read-intensive and mixed workloads, H-*Tucana* is more lightweight not only in CPU utilization but also in the amount of data read. Our modified B^{ε}–tree performs faster lookups than the LSM-trees used by HBase and Cassandra, obtaining significant improvement in throughput.

During the load phase (write intensive workloads) for



(a) 10 qualifiers (b) 1 qualifier

Figure 13: Number of cycles needed by H-*Tucana* for YCSB workloads.

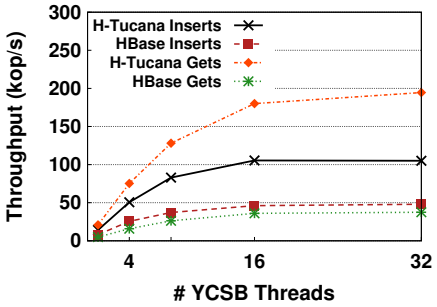


Figure 14: Scalability of H-*Tucana* and HBase with the small dataset.

the large dataset H-*Tucana* exhibits up 2.5× and 3.7× worse throughput than HBase and Cassandra, as shown in Figure 11b). Figure 12 shows that during the load phase, H-*Tucana* writes 264 GB and reads 69 GB, although the size of the dataset including metadata is 77.2 GB. This is not inherent to the design of *Tucana*, as shown by the results in Section 4.2, but rather due to `mmap`, as follows.

With `mmap` modified disk blocks are written to the device not only during *Tucana*'s commit operations, but also periodically, by the flush kernel threads when they are older than a threshold or when free memory shrinks below a threshold, using an LRU policy and `madvise` hints. We believe that due to the increased memory pressure in H-*Tucana* compared to *Tucana* due to the Java HBase front-end, `mmap` evicts not only log pages, but also leaf pages. This reduces the amount of I/Os that can be amortized for inserts due to the limited buffering in our B^ϵ–tree. To solve this problem, we need to (a) control better which pages are evicted by `mmap`, which will be effective up to roughly the 10-15% ratio of memory to SSD capacity (see Section 3.1), and (b) add buffering one level higher in the B^ϵ–tree. In the same figure, we notice that in run D phase, using the small dataset, we write more data than the other systems. This is because workload D inserts new key-value pairs and then searches for them. YCSB always searches for keys that exist in the database. In *Tucana* newly inserted keys appear in searches only after a commit operation. If a key is not found, we issue a commit operation to read it. These commit operations cause increased traffic to/from the device. However the other systems retrieve the new values directly from memory. In the large dataset case all systems write them to devices and all of them write about the same amount of data.

Figure 13 shows the cycles/op in H-*Tucana* to execute all the workloads with ten (default configuration) and with one qualifier. With ten qualifiers, write intensive workloads require on average 172K cycles/op and read intensive and mix workloads require on average 115K cycles/op. Workload E that performs scans uses more than 2.3M cycles/op (for retrieving 50 key-value pairs). Figure 13b shows that with a single key, all write-intensive, read-intensive, and mixed workloads require on average 27K cycles/op, whereas workload E requires 900K cycles/op. In more detail, we see that on average 40% of the time is used by the HBase component in H-*Tucana*, 23% by *Tucana*, 33% by the system, and 5% by other processes.

Figure 14 shows the scalability for H-*Tucana* and HBase when increasing the number of YCSB threads at the client. We have not measured the scalability for Cassandra because it is not as competitive. We use the small dataset to avoid accesses to the storage device.

For gets, both systems scale up to 16 YCSB threads. At this point CPU utilization for H-*Tucana* on the server side is 52% and for HBase is 79%, while H-*Tucana* achieves higher throughput. In both cases there is a single thread that reaches 100% CPU utilization. We find that this server thread performs HBase network processing. For inserts, H-*Tucana* scales up to 16 YCSB threads and HBase scales up to 8 YCSB threads. In H-*Tucana*, server CPU utilization is 53%, whereas in HBase 63%. Similar to gets, a single server thread in the HBase front-end limits further scalability.

5 Related Work

B-trees are a prominent structure [4, 40, 41] with good asymptotic behavior for searches and range queries. However, B-trees do not amortize I/Os for inserts and exhibit performance degradation for range queries when they age [17]. This has led to the design of write optimized structures, such as LSM-trees [43], B^ϵ–trees [7], Fractal trees [44], binomial lists [5], and Fibonacci Arrays [46]. Most of these structures introduce some type of I/O amplification due to compactions. LSM-trees in particular are broadly used today by key-value stores, including LevelDB, RocksDB, HBase, and Cassandra [2, 10, 18, 22, 29]. We categorize related work as follows.

Reducing I/O amplification: WiscKey [32] is based on an LSM-tree and does not require indexes but rather relies on sorted containers and compactions. WiscKey removes the values from the LSM-tree and stores them in an external log. Each key contains a pointer to the corresponding value. This technique improves compactions, because it eliminates the need to sort values. Additionally, it reduces the number of levels in the tree and therefore, the total number of compactions required. *Tucana* is based on a B^{ε}–tree, which has better inherent behavior with respect to amplification. Furthermore, WiscKey inherits explicit I/Os and WAL-based recovery from LevelDB, while *Tucana* was designed to use `mmap` for I/O and CoW for persistency. WiscKey improves performance when values are large compared to keys. *Tucana* on the other hand is designed without particular requirements on sizes of values and keys.

bLSM [47] improves read amplification in LSM-trees with two additional techniques, enhanced use of Bloom filters and efficient scheduling of compactions. VT-Tree [48] tries to reduce I/O amplification by merging efficiently sorted segments of non-overlapping levels of the tree. LSM-trie [53] constructs a trie structure of LSM-trees, and uses a hash-based key-value item organization. B*e*trFS [26], which is based on Fractal Tree Indexes [44], introduces heuristics to reduce write amplification and uses indexes at the buffer level for efficient lookups. Another approach, typically used in distributed NoSQL stores, is to offload compaction to servers managing replicas [1, 21]. *Tucana* starts from a structure that does not require compactions at the expense of more random I/Os. In addition, *Tucana* tries to improve CPU efficiency, which has not been the target of these systems.

SSDs and NVM: FlashStore [14] is a key-value store which builds a storage hierarchy with memory, flash, and disk to provide efficient lookups. NVMKV [35, 36] exploits native FTL capabilities to eliminate write amplification. SkimpyStash [15] stores the key-value pairs in a log-structured manner on flash SSD, to reduce memory footprint. SILT [30] combines three basic structures, a hash, a log, and a sorted store to achieve low memory footprint and reduce read and write amplification. These systems are mainly based on hash structures so they are not able to support efficient prefix and range queries found in analytics. Mercury [20] is an in-memory key-value store that uses a chained hash table and targets real-time applications without scan operations. Masstree [34] uses a trie-like concatenation of B+-trees to handle variable size keys. Masstree does not amortize I/Os for insert operations and scans are challenging to support and inherently expensive with the proposed structure.

At the device level, Wang *et al.* [51] leverage the parallelism in SDF [42], an open-channel SSD whose internal channels can be directly accessed, by providing multi-threaded I/O accesses in the write traffic control policy of LevelDB. They examine the ability to support new operations and interface as is the case with Open Channel SSD [51]. *Tucana* uses the storage device as a black box and it works with off-the-shelf SSDs.

In-memory operation: Silo [49] is inspired by Masstree and it is an in-memory store that does not offer persistence and targets efficient network behavior. HERD [28] is an in-memory key-value cache that leverages RDMA features to deliver low latency and high throughput. Its design is based on MICA [31], an in-memory key-value store that uses a lossy associative index to map keys to pointers and stores the values in a circular log. *Tucana* supports persistence and sits below the network layer.

6 Conclusions

In this work we present *Tucana*, a key-value store that is designed for fast storage devices, such as SSDs, that reduces the gap between sequential and random I/O performance, especially under high degree of concurrency and relatively large I/Os (a few tens of KB). Unlike most key-value stores that use LSM-trees to optimize writes over slow HDDs, *Tucana* starts from a B^{ε}–tree approach to maintain the desired asymptotic properties for inserts. It is a full-feature key-value store that supports variable size keys and values, versions, arbitrary data set sizes, and persistence. The design of *Tucana* centers around three techniques to reduce overheads: copy-on-write, private allocation, and direct device management.

Our results show that *Tucana* is up to 9.2× more efficient in terms of CPU cycles/op for in-memory workloads and up to 7× for workloads that do not fit in memory. In addition, *Tucana* outperforms RocksDB for in memory workloads up to 7×, whereas for workloads that do not fit in memory both systems are limited by device I/O throughput. Also, H-*Tucana* is able to improve up to 8× the efficiency of HBase and on average 22× the efficiency of Cassandra.

7 Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Program through the CoherentPaaS (FP7-ICT-611068) and LeanBigData (FP7-ICT-619606) projects, and the NESUS COST programme Action IC1305. We are thankful to Manolis Marazakis for his helpful comments. Finally, we thank the anonymous reviewers and our shepherd Angela Demke Brown for their insightful comments.

References

[1] AHMAD, M. Y., AND KEMME, B. Compaction management in distributed key-value datastores. *Proceedings of the VLDB Endowment 8*, 8 (2015), 850–861.

[2] APACHE. Hbase. `https://hbase.apache.org/`. Accessed: May 23, 2016.

[3] AXBOE, J. Flexible I/O Tester. https://github.com/axboe.

[4] BAYER, R., AND MCCREIGHT, E. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[5] BENDER, M. A., FARACH-COLTON, M., FINEMAN, J. T., FOGEL, Y. R., KUSZMAUL, B. C., AND NELSON, J. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures* (New York, NY, USA, 2007), SPAA '07, ACM, pp. 81–92.

[6] BONWICK, J., AND MOORE, B. Zfs: The last word in file systems.

[7] BRODAL, G. S., AND FAGERBERG, R. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 546–554.

[8] BURNS, R., AND HINEMAN, W. A bit-parallel search algorithm for allocating free space. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on* (2001), pp. 302–310.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems 26*, 2 (June 2008), 4:1–4:26.

[10] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[11] CHEN, S., GIBBONS, P. B., AND MOWRY, T. C. Improving index performance through prefetching. *ACM SIGMOD Record 30*, 2 (2001), 235–246.

[12] CHODOROW, K. *MongoDB: The Definitive Guide*, second ed. O'Reilly Media, 5 2013.

[13] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[14] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: High throughput persistent key-value store. *Proccedings of the VLDB Endowment 3*, 1-2 (Sept. 2010), 1414–1425.

[15] DEBNATH, B., SENGUPTA, S., AND LI, J. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), pp. 25–36.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[17] ESMET, J., BENDER, M. A., FARACH-COLTON, M., AND KUSZMAUL, B. C. The tokufs streaming file system. In *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems* (Berkeley, CA, USA, 2012), HotStorage'12, USENIX Association, pp. 14–14.

[18] FACEBOOK. Rocksdb. `http://rocksdb.org/`. Accessed: May 23, 2016.

[19] FACEBOOK. Rocksdb performance benchmarks. `https://github.com/facebook/rocksdb/wiki/Performance-Benchmarks`, 2015.

[20] GANDHI, R., GUPTA, A., POVZNER, A., BELLUOMINI, W., AND KALDEWEY, T. Mercury: Bringing efficiency to key-value stores. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, ACM, pp. 6:1–6:6.

[21] GAREFALAKIS, P., PAPADOPOULOS, P., AND MAGOUTIS, K. Acazoo: A distributed key-value store based on replicated lsm-trees. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on* (2014), IEEE, pp. 211–220.

[22] GOOGLE. Leveldb. `http://leveldb.org/`. Accessed: May 23, 2016.

[23] GOOGLE. Leveldb benchmarks. `https://leveldb.googlecode.com/svn/trunk/doc/benchmark.html`, 2015.

[24] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 981–992.

[25] JANG, K., SHERRY, J., BALLANI, H., AND MONCASTER, T. Silo: Predictable message latency in the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 435–448.

[26] JANNEN, W., YUAN, J., ZHAN, Y., AKSHINTALA, A., ESMET, J., JIAO, Y., MITTAL, A., PANDEY, P., REDDY, P., WALSH, L., BENDER, M., FARACH-COLTON, M., JOHNSON, R., KUSZMAUL, B. C., AND PORTER, D. E. Betrfs: A right-optimized write-optimized file system. In *13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, Feb. 2015), USENIX Association, pp. 301–315.

[27] JENKINS, B. A hash function for hash Table lookup. `http://www.burtleburtle.net/bob/hash/doobs.html`, 2009.

[28] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 295–306.

[29] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[30] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 1–13.

[31] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. Mica: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), pp. 429–444.

[32] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Wisckey: Separating keys from values in ssd-conscious storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 133–148.

[33] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems* (2012), ACM, pp. 183–196.

[34] MAO, Y., KOHLER, E., AND MORRIS, R. T. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 183–196.

[35] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., AND RANGASWAMI, R. Nvmkv: A scalable, lightweight, ftl-aware key-value store. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (2015), pp. 207–219.

[36] MARMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. Nvmkv: A scalable and lightweight flash aware key-value store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, June 2014), USENIX Association.

[37] MARTINEZ-BAZAN, N., GOMEZ-VILLAMOR, S., AND ESCALE-CLAVERAS, F. Dex: A high-performance graph database management system. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on* (April 2011), pp. 124–127.

[38] MCKENNEY, P. E., APPAVOO, J., KLEEN, A., KRIEGER, O., RUSSELL, R., SARMA, D., AND SONI, M. Read-copy update. In *AUUG Conference Proceedings* (2001), AUUG, Inc., p. 175.

[39] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: Virtual disks for virtual machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (New York, NY, USA, 2008), Eurosys '08, ACM, pp. 41–54.

[40] MYSQL, A. Mysql, 2001.

[41] OLSON, M. A., BOSTIC, K., AND SELTZER, M. I. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track* (1999), pp. 183–191.

[42] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. Sdf: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)* (2014), pp. 471–484.

[43] ONEIL, P., CHENG, E., GAWLICK, D., AND ONEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[44] PERCONA, L. Perconaft. `https://github.com/percona/PerconaFT`. Accessed: May 23, 2016.

[45] REN, J. Ycsb-c. `https://github.com/basicthinker/YCSB-C`, 2015.

[46] SANTRY, D., AND VORUGANTI, K. Violet: A storage stack for iops/capacity bifurcated storage environments. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, 2014), USENIX Association, pp. 13–24.

[47] SEARS, R., AND RAMAKRISHNAN, R. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, ACM, pp. 217–228.

[48] SHETTY, P. J., SPILLANE, R. P., MALPANI, R. R., ANDREWS, B., SEYSTER, J., AND ZADOK, E. Building workload-independent storage with vt-trees. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 17–30.

[49] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 18–32.

[50] VMWARE, I. Vmware virtual san 6.1 product datasheet. `https://www.vmware.com/files/pdf/products/vsan/VMware_Virtual_SAN_Datasheet.pdf`. Accessed: May 23, 2016.

[51] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 16.

[52] WEBBER, J. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity* (2012), ACM, pp. 217–218.

[53] WU, X., XU, Y., SHAO, Z., AND JIANG, S. Lsm-trie: An lsm-tree-based ultra-large key-value store for small data items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 71–82.

Samsara: Efficient Deterministic Replay in Multiprocessor Environments with Hardware Virtualization Extensions

Shiru Ren[1], Le Tan[1], Chunqi Li[1], Zhen Xiao[1], and Weijia Song[2]

[1]*Department of Computer Science, Peking University*
[2]*Department of Computer Science, Cornell University*

Abstract

Deterministic replay, which provides the ability to travel backward in time and reconstruct the past execution flow of a multiprocessor system, has many prominent applications. Prior research in this area can be classified into two categories: hardware-only schemes and software-only schemes. While hardware-only schemes deliver high performance, they require significant modifications to the existing hardware which makes them difficult to deploy in real systems. In contrast, software-only schemes work on commodity hardware, but suffer from excessive performance overhead and huge logs caused by tracing every single memory access in the software layer.

In this paper, we present the design and implementation of a novel system, Samsara, which uses the hardware-assisted virtualization (HAV) extensions to achieve efficient and practical deterministic replay without requiring any hardware modification. Unlike prior software schemes which trace every single memory access to record interleaving, Samsara leverages the HAV extensions on commodity processors to track the read-set and write-set for implementing a chunk-based recording scheme in software. By doing so, we avoid all memory access detections, which is a major source of overhead in prior works. We implement and evaluate our system in KVM on commodity Intel Haswell processor. Evaluation results show that compared with prior software-only schemes, Samsara significantly reduces the log file size to 1/70th on average, and further reduces the recording overhead from about 10×, reported by state-of-the-art works, to 2.3× on average.

1 Introduction

Modern multiprocessor architectures are inherently non-deterministic: they cannot be expected to reproduce the past execution flow exactly, even when supplied with the same inputs. The lack of reproducibility complicates software debugging, security analysis, and fault-tolerance. It greatly restricts the development of parallel programming.

Deterministic replay helps reconstruct non-deterministic processor executions. It is extensively used in a wide range of applications. For software debugging, it is the most effective way to reproduce bugs, which helps the programmer understand the causes of the bug [1, 33]. For security analysis, it can help the system administrator analyze the intrusions and investigate whether a specific vulnerability was exploited in a previous execution [17, 18, 7, 11, 37]. For fault-tolerance, it provides the ability to replicate the computation on processors for building the hot-standby system or data recovery [6, 43, 42, 32].

In the multiprocessor environment, memory accesses from multiple processors to a shared memory object may interleave in any arbitrary order, which become a significant source of non-determinism and pose a formidable challenge to deterministic replay. To address this problem, most of the existing research focuses on how to record and replay the memory access interleaving using either a pure hardware scheme or a pure software scheme.

Hardware-only schemes record memory access interleaving efficiently by embedding special hardware components into the processors and redesigning the cache coherence protocol to identify the coherence messages among processors [38, 22, 21, 15, 9, 29, 14, 35, 23, 30]. The advantage of such a scheme is that it allows efficient recording of memory access interleaving in a multiprocessor environment. On the down side, it requires extensive modifications to the existing hardware, which significantly increases the complexity of the circuits and makes them largely impractical in real systems.

In contrast, software-only schemes achieve deterministic replay on the existing hardware by modifying the OS, the compiler, the runtime libraries or the virtual machine manager (VMM) [19, 13, 8, 34, 33, 34, 25, 2,

26, 20, 41, 4]. Among them, virtualization-based deterministic replay is one of the most promising approaches which provides full-system level replay by leveraging the concurrent-read, exclusive-write (CREW) protocol to serialize and log the total order of the memory access interleaving [13, 8, 27]. While these schemes are flexible, extensible, and user-friendly, they suffer serious performance overhead (about 10× compared to the native execution) and generate huge logs (approximately 1 MB/s on a four core processor after compression). The poor performance can be ascribed to the numerous page fault VM exits led by tracing every single memory access in the software layer.

To summarize, it is inherently difficult to record memory access interleaving efficiently by software alone without proper hardware support. Although there is no commodity processor with dedicated hardware-based record and replay capability, some advanced hardware features in these processors are available to boost the performance of the software-based deterministic replay systems. Therefore, we argue that the software scheme can be a viable approach in the foreseeable future if it can take advantages of advanced hardware features.

In this paper, the main goal is to implement a software approach that can take full advantages of the latest hardware features in commodity processors to record and replay memory access interleaving efficiently without introducing any hardware modifications. The emergence of hardware-assisted virtualization (HAV) provides the possibility to meet our requirements. Although HAV cannot be used for tracing memory access interleaving directly, we have found a novel use of it to track the read-set and write-set, and bypass the time-consuming process in traditional software schemes. Specifically, we abandon the inefficient CREW protocol that records the dependence between individual instructions, and instead use a chunk-based strategy that records processors' execution as a series of chunks. By doing so, we avoid all memory access detections, and instead obtain each chunk's read-set and write-set by retrieving the accessed and the dirty flags of the extended page table (EPT). These read and write sets are used to determine whether a chunk could be committed, and the determinism is ensured by recording the chunk size and the commit order. To further improve the system performance, we propose a decentralized three-phase commit protocol, which significantly reduces the performance overhead by allowing chunk commits in parallel while still ensuring serializability.

We implement our prototype, Samsara, which, to the best of our knowledge, is the first software-based deterministic replay system that can record and replay memory access interleaving efficiently by leveraging the HAV extensions on commodity processors. Experimental results show that compared with prior software schemes based on the CREW protocol, Samsara reduces the log file size to 1/70th on average (from 0.22MB/core/second to 0.003MB/core/second) and reduces the recording overhead from about 10× to 2.3× compared to the native execution.

Our main contributions are as follows:

- We present a software-based deterministic replay system that can record and replay memory access interleaving efficiently by leveraging the HAV extensions. It improves the recording performance dramatically with a log size much smaller than all prior approaches.
- We design a decentralized three-phase commit protocol, which further improves the performance by enabling the chunk commit in parallel while ensuring serializability.
- We build and evaluate our system in KVM on Intel Haswell processor, and we plan to open-source our system to the community.

The rest of the paper is organized as follows. Section 2 describes the general architecture and shows how Samsara achieves deterministic replay. Section 3 illustrates how to record and replay the memory access interleaving. Section 4 presents the optimization and the implementation details. We evaluate Samsara in section 5. Section 6 reviews related work and section 7 concludes the paper.

2 System Overview

In this section, we present the system overview of Samsara. We first outline the overall architecture of Samsara. Then, we discuss how it records and replays all non-deterministic events.

2.1 System Architecture

Samsara implements the deterministic replay as an extension to VMM, which has access to the entire virtual machine and can take full advantage of the HAV extensions, as illustrated in Figure 1. The architecture of Samsara consists of four principal components, namely, the Controller, the record and replay component, the DMA recorder, and the log record daemon as shown in orange boxes in the figure. The controller is in charge of all policy enforcement. It provides a control interface to users, manages the record and replay component in KVM, and is in charge of the log transfer. The record and replay component acts as a part of VMM working in the kernel space being responsible for recording and replaying all

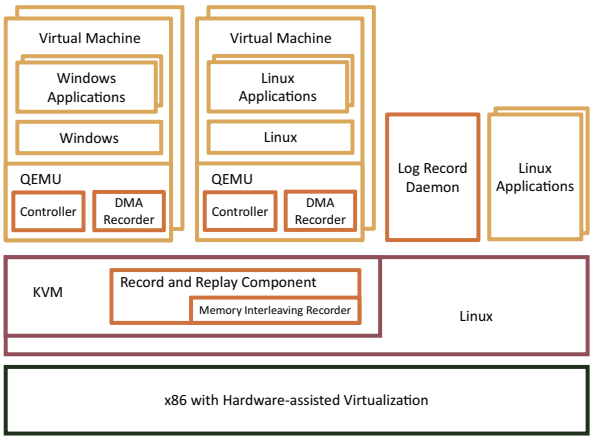


Figure 1: Architecture overview.

non-deterministic events, especially the memory access interleaving. The DMA recorder records the contents of DMA events as part of QEMU. Finally, we optimize the performance of logging by utilizing a user-space log record daemon. It runs as a background process that supports loading and storing log files.

Samsara implements deterministic replay by first logging all non-deterministic events during the recording phase and then reproducing these events during the replay phase. Before recording, the controller initializes a snapshot of the whole VM states. Then all non-deterministic events and the exact points in the instruction stream where these events occurred will be logged by the record and replay component during recording. Meanwhile, it transfers these log data to the userspace log record daemon, which is responsible for the persistent storage and the management of the logs. The replay phase is initialized by loading the snapshot to restore all VM states. During replay, the execution of the virtual processors is controlled by the record and replay component which ignores all external events. Instead each recorded event will be injected at the exact same point as in the recorded execution.

2.2 Record and Replay Non-deterministic Events

Non-deterministic events fall into three categories: synchronous, asynchronous, and compound. The following illustrates what events will be recorded and how recording and replaying is done in our system.

Synchronous Events. These events are handled immediately by the VM when they occur. They always take place at the exact same point where they appear in the instruction stream, such as I/O events and RDTSC instructions. The key observation is that they will be triggered by the associated instructions at the fixed point if all previous events are properly injected. Therefore, we just record the contents of these events. During replay, we merely inject logged data to where the I/O (or RDTSC) instruction is trapped into the VMM.

Asynchronous Events. These events are triggered by external devices, such as external interrupts, so they may appear at any arbitrary time from the point of view of the VM. Their impact to the state of the system is deterministic, but the timing of their occurrences is not. To replay them, all such events must be identified with a three-tuple timestamp (including program counter, branch counter, and the value of ECX) like the approach in ReVirt [12]. The first two are used to uniquely identify the instruction where the event appears in the instruction stream. However, the x86 architecture introduces the REP prefixes to repeat a string instruction the number of times specified in the ECX. Therefore, we also need to log the value of ECX which stores how many iterations remain at the time of this event takes place [12]. During replay, we leverage a hardware performance counter to guarantee that the VM stops at the recorded timestamp to inject them.

Compound Events. These events are non-deterministic in both their timing and their impact on the system. DMA is an example of such events: the completion of a DMA operation is notified by an interrupt which is asynchronous, and the data copy process is initialized by a series of I/O instructions which are synchronous. Hence, it is necessary to record both the completion time and the content of a DMA event.

Memory Access Interleaving. In the multiprocessor environment, memory accesses from multiple processors to a shared memory object may interleave in any arbitrary order, which become a significant source of non-determinism. More specifically, if two instructions both access the same memory object and at least one of them is write, then the access order of these two instructions should be recorded during the recording phase. Unfortunately, the number of such events is orders of magnitude larger than all the other non-deterministic events combined. Therefore, how to record and replay these events is the most challenging problem in a replay system.

3 Record and Replay Memory Access Interleaving with HAV Extensions

How to record and replay memory access interleaving efficiently is the most significant challenge we face during the design and implementation of Samsara. In this section, we describe on how Samsara uses HAV extensions to overcome this challenge. Firstly, we show the specific design of our chunk-based strategy, then we discuss two

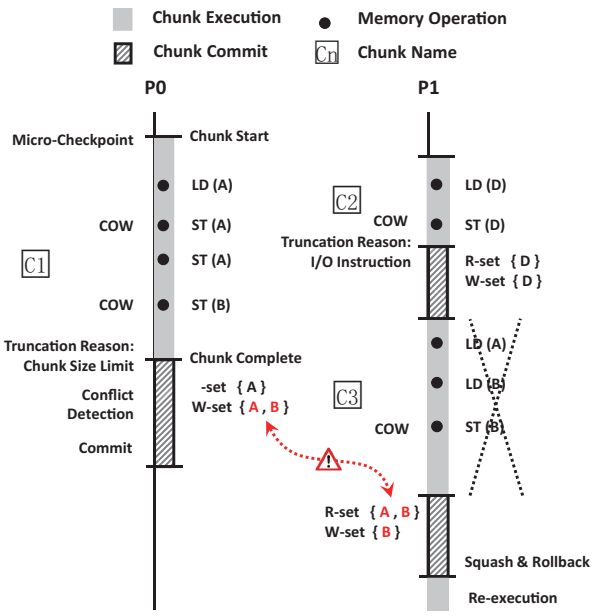


Figure 2: The execution flow of our chunk-based approach.

technical issues when implementing this strategy in software: how to obtain the read-set and write-set efficiently and how to reduce the commit overhead. Finally, we give a brief description on how to replay the memory access interleaving in Samsara.

3.1 Chunk-based Strategy

Previous software-only schemes leverage CREW protocol to serialize and log the total order of the memory access interleaving [19], which produces huge log size and excessive performance overhead because every single memory access needs to be checked for logging before execution. Therefore, chunk-based approach has been proposed on the hardware-based replay system to reduce the log size [21]. In this approach, each processor executes instructions grouped into chunks. Thus, it just needs to record the total order of chunks. However, this approach is not directly applicable to a software-only replay system, because tracing every single memory access to obtain the read-set and write-set during chunk execution in software will still be as time-consuming as directly logging the memory access interleaving itself. To eliminate this performance overhead, we find HAV extension extremely useful. Instead of tracing every single memory access, HAV offers a fast shortcut to track the read-set and write-set, which can be used to implement the chunk-based approach in software layer.

To implement a chunk-based recording scheme, we need to divide the execution of virtual processors into a series of chunks. In our system, a chunk is defined as a finite sequence of machine instructions. Similarly to the database transaction, chunk execution must satisfy the atomicity and serializability requirements. Atomicity requires that the execution of each chunk must be "all or nothing". Serializability requires that the concurrent execution of chunks have to result in the same system state as if these chunks were executed serially.

To enforce serializability, firstly, we must guarantee no update within a chunk is visible to other chunks until it commits. Thus, on the first write to each memory page within a chunk, we create a local copy on which to perform the modification by leveraging copy-on-write (COW) strategy. When a chunk completes execution, it either gets committed, copying all local data back to the shared memory, or gets squashed, discarding all local copies. Moreover, an efficient conflict detection strategy is necessary to enforce serializability. Particularly, an executing chunk must be squashed and re-executed when its accessed memory pages have been modified by a newly committed chunk. To optimize recording performance, we leverage lazy conflict detection. Namely, we defer detection until chunk completion. When a chunk completes, we obtain the read-set and write-set (R&W-set) of this chunk. We intersect all write-sets of other concurrent chunks with this R&W-set afterwards. If the intersection is not empty, which means there are collisions, then this chunk must be squashed and re-executed. Note that the write-write conflict must be detected even if there is no read in these chunks. Specifically, the conflict detection is implemented at the page-level granularity, therefore any attempts to make the write-conflicting chunks serial may overwrite uncommitted data and cause a lost update. Finally, there are certain instructions that may violate atomicity because they lead to externally observable behaviors (e.g., I/O instructions may modify device status and control activities on a device). Once any of such instructions has been executed in a chunk, this chunk could no longer be rolled back. Therefore, we truncate a chunk when any of such instructions is encountered. Then the execution of such instructions must be deferred until this chunk can be committed.

Figure 2 illustrates the execution flow of our chunk-based approach. First, we make a micro-checkpoint of the status of a virtual processor at the beginning of each chunk. During chunk execution, the first write to each memory page will trigger a COW operation that creates a local copy. All the following modifications to this page will be performed on this copy until chunk completion. A currently running chunk will be truncated when an I/O operation occurs or if the number of instructions executed within this chunk reaches the size limit. When a chunk completes, we obtain its R&W-set. Then the con-

flict detection is done by intersecting its own R&W-set with all W-sets of other chunks which just committed during this chunk execution. If the intersection is empty (as C1 or C2 in Figure 2), this chunk can be committed. Finally, we record the chunk size and the commit order which together are used to ensure that this chunk will be properly reconstructed during replay. Otherwise (as C3 in Figure 2), all local copies will be discarded and we rollback the status of the virtual processor with the micro-checkpoint we made at the beginning and re-execute this chunk.

In our design, there are two major challenges: 1) how to obtain the R&W-set (section 3.2); 2) how to commit the chunks in parallel while ensuring serializability (section 3.3).

3.2 Obtain R&W-set Efficiently via HAV

The biggest challenge in the implementation of a chunk-based scheme in software is how to obtain the R&W-set efficiently. Hardware-based schemes achieve this by tracing each cache coherence protocol message. However, doing so in software-only schemes will result in serious performance degradation.

Fortunately, the emergence of HAV provides the possibility to reduce this overhead dramatically. HAV extensions enable efficient full-system virtualization utilizing the help from hardware capabilities. Take Intel Virtualization Technology (Intel VT) as an example. It provides hardware support for simplifying x86 processor virtualization. The EPT that provided in HAV is a hardware-assisted address translation technology, which can be used to avoid the overhead associated with software managed shadow page tables. Intel Haswell microarchitecture also introduces the accessed and dirty flags for EPT, which enables hardware to detect which page has been accessed or updated during execution. More specifically, whenever the processor uses an EPT entry as part of the address translation, it sets the accessed flag in that entry. In addition, whenever there is a write to a guest-physical address, the dirty flag in the corresponding entry will be set. Therefore, by utilizing these hardware features, we can obtain the R&W-set by gathering all leaf entries where the accessed or the dirty flag is set, which can be archived by a simple EPT traversal.

Moreover, the tree-based design of EPT makes it possible to further improve performance. EPT uses a hierarchical, tree-based design which allows the subtrees corresponding to some unused part of the memory to be absent. A similar feature is also present for the accessed and the dirty flags. For instance, if the accessed flag of one internal entry is 0, then the accessed flags of all page entries in its subtrees are definitely 0. Hence, it is not necessary to traverse these subtrees. In practice, due to locality of reference, the access locations of most chunks are adjacent. Thus, we usually just need to traverse a tiny part of EPT, which incurs negligible overhead.

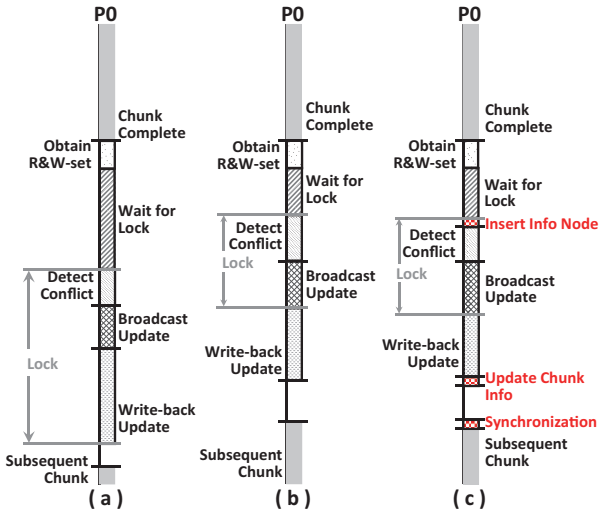


Figure 3: General design of decentralized three-phase commit protocol: a) chunk timeline of a naïve design, b) moving update write-back operation out of the synchronized block, and c) a design of decentralized three-phase commit protocol.

3.3 A Decentralized Three-Phase Commit Protocol

Apart from obtaining the R&W-set, chunk commit is another time-consuming process. In this section, we discuss how to optimize this part using a decentralized three-phase commit protocol.

Some hardware-based solutions add a centralized arbiter module to processors to ensure that one chunk gets committed at a time, without overlapping [21]. However, when it comes to software-only schemes, an arbiter will be slow. Thus, we propose a decentralized commit protocol to perform chunk commit efficiently.

The chunk commit process includes at least three steps in our design: 1) conflict detection that determines whether this chunk can be committed, 2) update broadcast that notifies other processors which memory pages are modified, 3) update write-back that copies all updates back to shared memory. A naïve design of the decentralized commit protocol is shown in Figure 3 a). Without a centralized arbiter, we leverage a system-wide lock to enforce serializability. Each virtual processor maintains three bitmaps: an access bitmap, a dirty bitmap, and a conflict bitmap. The first two bitmaps help mark which memory pages were accessed or updated dur-

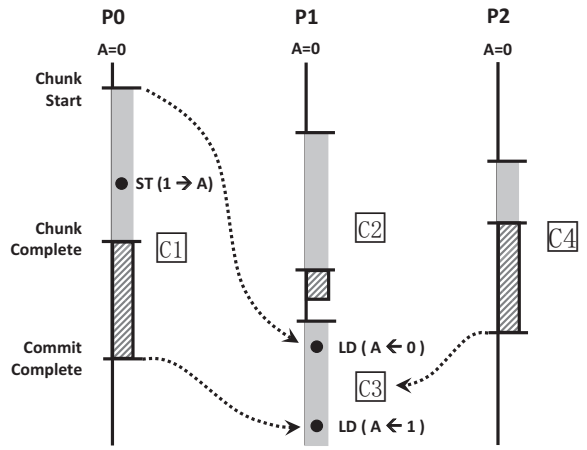


Figure 4: An example of out-of-order commit.

ing the chunk execution (same as the R&W-set). Each bit in the conflict bitmap indicates whether its corresponding memory page was updated by other committing chunks. To detect conflict, we just need to intersect the first two bitmaps with the last one. If the intersection is empty which means this chunk can be committed, this virtual processor broadcasts its W-set to notify others which memory pages have been modified by performing a bitwise-OR operation between the other virtual processors' conflict bitmaps and its own dirty bitmap. Then it copies its local data back to the shared memory. Finally, it clears its three bitmaps before the succeeding chunk starts. This whole commit process is performed while holding this lock.

However, lock contention turns out to cause significant performance overhead. In our experiments, it contributes to nearly 40% of the time spent on committing the chunks. To address this issue, we redesign the commit process to reduce the lock granularity. We observe that the write-back operation involves serious performance degradation due to lots of page copies, and all these pages committed concurrently by different chunks have no intersection, which is already guaranteed by conflict detection. Based on this observation, we move this operation out of the synchronized block to reduce the lock granularity, as shown in Figure 3 b). This not only reduces the cost of the locking operation substantially, but also increases parallelism because multiple chunks can now commit concurrently.

However, one side effect of this design is that chunks may get committed out-of-order, thereby violating serializability. One example is shown in Figure 4. C1 writes A, then finishes its execution first and starts to commit. Then, C2 starts committing as well and finishes before C1. Meanwhile C3 starts to execute and happens to read A immediately. Unfortunately, C1 may not accomplish its commit process in such a short period, thus C3 fetches the obsolete value of A. Suppose C3 reads A again and gets a new value after C1 completes its commit. Then C3 gets two different values of the same memory object, which violates serializability. To maintain serializability, we need to guarantee that before starting C3, P1 waits until all the other chunks which start committing prior to the commit point of C2 (e.g., C1 and C4) complete their commit.

We develop a decentralized three-phase commit protocol to support parallel commit while ensuring serializability. To eradicate out-of-order commits, we introduce a global linked list, *commit_order_list*, which maintains the order and information of each current committing chunk. Each node of this list contains a commit flag field to indicate whether the corresponding chunk has completed its commit process. Moreover, this list is kept sorted by the commit order of its corresponding chunk. A lock is used to prevent multiple chunks from updating this list concurrently. This protocol consists of three phases as shown in Figure 3 c):

1) The pre-commit phase: In this phase, each processor must register its commit information by inserting an info node at the end of the *commit_order_list*. The commit flag of this info node will be initialized to 0, which means this chunk is about to be committed.

2) The commit phase: In this phase, the memory pages updated by this chunk will be committed (i.e., written back to shared memory). Then the processor must set the commit flag of its info node to 1 at the end of this phase, which means it has completed its commit process. Chunks can commit in parallel in this phase, because pages committed by different chunks have no intersection.

3) The synchronization phase: In this phase, this virtual processor is blocked until all the other chunks which start committing prior to the commit point of its preceding chunk have completed their commit. To enforce this, it needs to check all commit flags of those chunk info nodes which are ahead of its own node. If at least one flag is 0, then this processor must be blocked. Otherwise, the processor removes its own info node from the *commit_order_list* and begins executing the next chunk. In practice, this blocking almost never happens, because a virtual processor tends to exit to QEMU to emulate device operations before executing the next chunk, which happens to provide sufficient time for other chunks to complete their commit.

This design noticeably improves performance via reducing the lock granularity. In brief, only the conflict de-

tection and the update broadcast operation are protected by a system-wide lock. Furthermore, It also reduces the time spent on waiting for the lock, because the shorter the time a chunk holds a lock, the lower the probability that other chunks requesting it have to wait is. The most important characteristic is that this protocol can satisfy the serializability requirement because it strictly guarantees that the processor starting to commit a chunk first will execute the subsequent chunk preferentially. The following of this section presents a formal proof on how our decentralized three-phase commit protocol ensures serializability.

Assume for the sake of contradiction that it does not guarantee serializability. Then there exists a set of chunks $C_0, C_1 \ldots C_{n-1}$ which obey our three-phase commit protocol and produce a non-serializable schedule. In order to know whether this chunk schedule is serializable or not, we can draw a precedence graph. This is a graph in which the vertices are the committed chunks and the edges are the dependencies between these committed chunks. A dependence $C_i \rightarrow C_j$ exists only if one of the following is true: 1) C_i executes $Store(X)$ before C_j executes $Load(X)$; 2) C_i executes $Load(X)$ before C_j executes $Store(X)$; 3) C_i executes $Store(X)$ before C_j executes $Store(X)$.

A non-serializable chunk schedule implies a cycle in this graph, and we will prove that our commit protocol cannot produce such a cycle. Assume that a cycle exists in the precedence graph like this: $C_0 \rightarrow C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_{n-1} \rightarrow C_0$, for each chunk C_i, we define T_i to be the time when C_i has been committed, and the corresponding processor begins executing its next chunk C_{i+1}. Then for chunks such that $C_i \rightarrow C_j$, $T_i < T_j$. This is because the *commit_order_list* maintains the total order of these current committing chunks on all processors, and the three-phase commit protocol guarantees that all chunks will be processed in FIFO order. Specifically, The pre-commit phase guarantees that the chunk will be inserted in the *commit_order_list* in execution order, and the synchronization phase guarantees that the chunk will be blocked until all the other chunks which start committing prior to it have completed their commits. Moreover, the conflict detection ensures that an executing chunk will be squashed and re-executed later when there are collisions between it and a newly committed chunk, therefore, will not affect the commit order. Then for this cycle, we have: $T_0 < T_1 < T_2 < \ldots < T_{n-1} < T_0$, which is a manifest contradiction. Hence, our three-phase commit protocol can ensure serializability.

3.4 Replay Memory Access Interleaving

It is relatively simple and efficient to replay memory access interleaving under a chunk-base strategy. Unlike the CREW protocol which must restrict every single memory access to reconstruct the recorded memory access interleaving, we just need to make sure that all chunks will be re-built properly and executed in the original order. In other words, our replay strategy is more coarse-grained.

When we design the replay mechanism of Samsara, a design goal is to maintain the same parallelism as the recoding phase. Since the atomicity and the serializability have already been guaranteed in recording phase, both the conflict detection and the update broadcast operations are no longer required during replay. We just need to ensure that all the preceding chunks have been committed successfully before the current chunk starts. More specifically, during replay, the processors generate chunks according to the order established by the chunk commit log. Then they use the chunk size in that log to determine when they need to truncate these chunks. Here, we use the same approach as above to confirm that a chunk can be truncated at the recorded timestamp. During chunk execution, the COW operation is also required to guarantee that the other concurrently executing chunks will not access the latest data updated by this chunk. To ensure chunk commit in the original order, we will block the commit of a chunk until all the preceding chunks have been committed successfully.

4 Optimizations and Implementation Details

This section describes several optimizations for our chunk-based strategy to improve the recording performance and some implementation details of Samsara.

4.1 Caching Local Copies

In our chunk-based strategy, a copy-on-write (COW) operation will be triggered to create a local copy on the first write to each memory page within a chunk. In the original design, these local copies will be destroyed at the end of this chunk. However, we find that these COW operations can cause a significant amount of performance overhead, especially when recording computation intensive applications.

By analyzing the memory access patterns, we observe that the write accesses of successive chunks exhibit great temporal locality with a history-similar pattern, which means they incline to access roughly the same set of pages. Particularly, when a rollback occurs, the re-executed chunk will follow a similar instruction flow and access the exact same set of pages in most instances.

Based on this observation, we decide to retain local copies at the end of each chunk and use them as a cache of hot pages. By doing so, when a processor modifies a page which already has a copy in the local cache, it

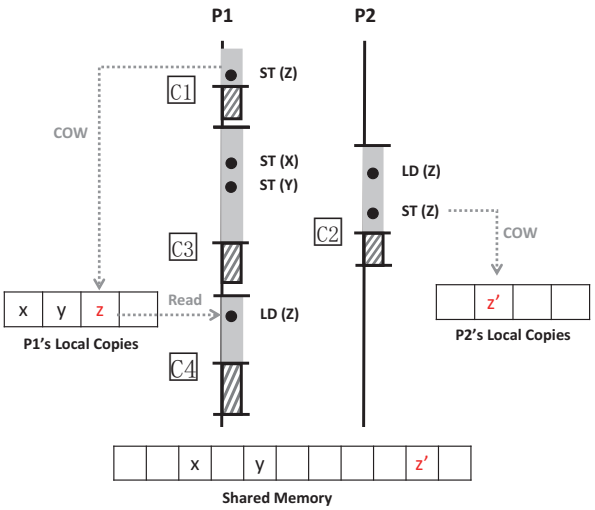


Figure 5: An example of reading outdated data from local copies.

acts just like it does in the unmodified VM with hardware acceleration, and no other operations will be necessary.

However, this design may cause chunks to read outdated data. One example is shown in Figure 5: chunk C4 reads z from its local cache, and meanwhile this page is modified to z' by another committed chunk C2 and copied back to the shared memory. This does not cause any collision, but unfortunately, chunk C4 reads the outdated data z.

These outdated copies can be simply detected by checking the corresponding bit in the conflict bitmap for each local copy. However, the crucial issue remains as how to deal with these outdated copies. We can either update local copies with the latest data in the shared memory or simply discard these outdated copies which have been modified by other committed chunks. These two strategies both have their own advantages and shortcomings: the former reduces the number of COW operations but leads to relatively high overhead due to frequent memory copy operations, while the latter avoids this overhead but still retains some COW operations. We combine the merits of these two strategies as follows: we update outdated copies when a rollback occurs, and discard them when a chunk is committed.

This optimization is essentially equivalent to adding a local cache to buffer the hot pages which are modified by successive chunks. In the current implementation, we limit this cache to a fixed size (0.1% of the main memory size) with a modified LRU replacement policy.

4.2 Adaptive Chunk Size

The chunk size is also a critical factor to the performance of the replay system. If the chunk size is too small, its execution time will not be long enough to amortize the cost of a chunk commit. On the other hand, if the chunk size is too large, the corresponding processor may experience repeated rollbacks due to the increased risk of collision during its commitment, which will eventually cause starvation. Moreover, different applications or even different execution regions in the same application can exhibit varying memory access patterns, which makes it difficult to seek the sweet spot for chunk size, because the optimal size might not be a constant, but rather change during execution. Therefore, one of the major challenges in our implementations is how to adjust chunk size adaptively to achieve a good balance.

Therefore, we propose an adaptive additive-increase/multiplicative-decrease (AIMD) algorithm to adjust the chunk size dynamically during runtime. In this algorithm, a processor will increase its chunk size by a fixed amount after each successful commit to probe for longer execution time. When collision is detected, the processor decreases its chunk size by a multiplicative factor. The idea is similar to the feedback control algorithm in TCP congestion avoidance [10]. The decrease must be multiplicative because it is the only effectual way to ensure that at every step the fairness either increases or stays the same [10]. In a nutshell, when a conflict takes place, this adaptive AIMD algorithm ensures that all processors will quickly converge to use equal chunk size. Therefore, each chunk has same probability to be committed or squashed. One limitation of this algorithm is that it is less effective for some I/O intensive workloads due to the frequent chunk truncations caused by the large number of concurrent I/O requests.

4.3 Double Buffering

In our decentralized commit protocol, the conflict detection and the update broadcast operation are both protected by a system-wide lock to enforce the serialization requirement. Since the conflict bitmap will be modified by other chunks due to the update broadcast operation, while being read by its own chunk for the conflict detection, it has to acquire this system-wide lock whenever one chunk try to set the corresponding bits in other chunks' conflict bitmap to broadcast its updates. Similarly, it has to wait for this lock to be released whenever its own chunk needs to read this bitmap for conflict detection.

Double buffering mitigates this problem and can further increase parallelism. Instead of using a single

bitmap, we use two bitmaps simultaneously to implement double buffering. One of them serves as a write bitmap and the other as the read bitmap. Both bitmaps can be accessed at any time. By doing so, we avoid locking the bitmap while reading and writing to it. We switch these two bitmaps when the succeeding chunk starts its conflict detection, so it can read the bitmap directly when other chunks are free to set this bitmap simultaneously. In our design, only this switch operation is protected by a lock, and neither bitmap requires any locking at all.

5 Evaluation

This section discusses our evaluation of Samsara. We first illustrate the experimental setup and our workloads. Then we evaluate different aspects of Samsara and compare it with a CREW approach.

5.1 Experimental Setup

All the experiments are conducted on a Dell Precision T1700 Workstation with a 4-core Intel Core i7-4790 processor (running at 3.6GHz, with 256KB L1, 1MB private L2 and 8MB shared L3 cache) running Ubuntu 12.04 with Linux kernel version 3.11.0 and QEMU-1.2.2. The host machine has 12GB memory. The Guest OS is an Ubuntu 14.04 with Linux kernel version 3.13.1.

5.2 Workloads

To evaluate our system on a wide range of applications, we choose two sets of benchmarks that represent very different characteristics, including both computation intensive and I/O intensive applications.

The first set includes eight computation intensive applications chosen from PARSEC and SPLASH-2 benchmark suites (four from each): blackscholes, bodytrack, raytrace, and swaptions form PARSEC [5]; radiosity, water_nsquared, water_spatial, and barnes from SPLASH-2 [36]. We choose both PARSEC and SPLASH-2 suites because each of them has its own merits, and no single benchmark can represent the characteristics of all types of applications. PARSEC is a well-studied benchmark suite composed of emerging multithreaded programs from a broad range of application domains. In contrast, SPLASH-2 is composed mainly of high-performance computing programs which are commonly used for scientific computation on distributed shared-address-space multiprocessors. These eight applications come from different areas of computing and are chosen because they exhibit diverse characteristics and represent the different worst-case applications due to the burdensome shared memory accesses.

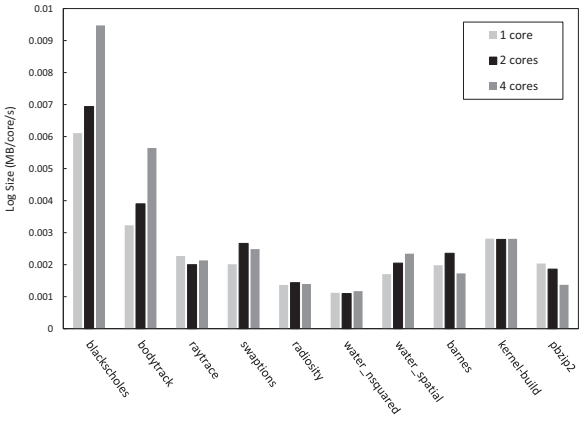


Figure 6: Log size produced by Samsara during recording (compressed with gzip).

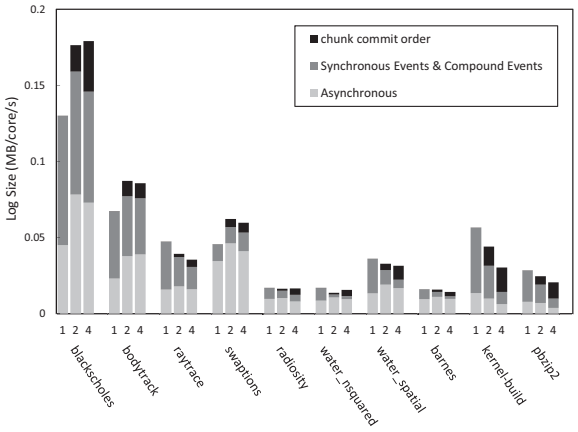


Figure 7: The proportion of each type of non-deterministic events in a log file (without compression).

Although there are applications in the first set that perform certain amount of I/O operations, most of them are disk read only. In the other set of benchmarks, we select two more I/O intensive applications (kernel-build and pbzip2) to further evaluate how well Samsara handle I/O operations. Kernel-build is a parallel build of the Linux kernel version 3.13.1 with the default configuration. In order to achieve maximum degree of parallelism we use the *-j* option of make. Usually, *make -j n+1* produces a relatively high performance on a VM with *n* virtual processors. This is because the extra process makes it possible to fully utilize the processors during network delays and general I/O accesses such as loading and saving files to disk [13]. Pbzip2 is a parallel file compressor which uses pthreads. We use pbzip2 to decompress a 111MB Linux-kernel source file.

5.3 Log Size

Log size is an important consideration of the replay systems. Usually, recording non-deterministic events will generate huge space overhead which limits the duration of the recording. The log size of some prior works is approximately 2 MB/1GHz-processor/s [38]. Some can support only a few seconds' recording which is difficult to satisfy long-term recording needs [38].

Experiment results show that Samsara produces a much smaller log size which is orders of magnitude smaller than the ones reported by prior work in software-based schemes, and even smaller than some reported in hardware-based schemes. Figure 6 shows the compressed log sizes generated by each core for all the applications. The experiments indicate that Samsara generates logs at an average rate of 0.0027 MB/core/s and 0.0031 MB/core/s for recording two and four cores, respectively. For comparison, the average log size with a single core, which does not need to record memory interleaving, is 0.0024 MB/s.

To compare the log size of Samsara and the previous software or hardware approaches, this experiment was designed to be as similar as possible to the ones in the previous papers. SMP-ReVirt generates logs at an average rate of 0.18MB/core/s when recording the workloads in SPLASH-2 and kernel-build on two dual-core Xeons [13]. DeLorean generates logs at an average rate of 0.03MB/core/s when recording the workloads in SPLASH-2 on eight simulated processors [21].

We achieve a significant reduction in the log size because the size of the chunk commit log is practically negligible compared with other non-deterministic events. Figure 7 illustrates the proportions of each type of non-deterministic events in each log file. In most workloads, the interleaving log represents a small fraction of the whole log (approximately 9.36% with 2 cores and 19.31% with 4 cores). For the I/O intensive applications, this proportion is higher, because the large number of concurrent I/O requests leads to more chunk truncations.

Another reason is we avoid recording all disk reads. In Samsara, we use QEMU's qcow2 (QEMU Copy On Write) disk format to create a write protected base image and an overlay image on top of it to perform disk modifications during recording and replay. By doing so, we can present the same disk view for replay without logging any disk reads or creating another copy of the whole disk image.

In summary, the use of chunk-based strategy makes it possible to significantly reduces the log file size by 98.6% compared to the previous software-only schemes. The log size in our system is even smaller than the ones reported in hardware-based solutions, since we can further reduce the log size via increasing the chunk size which is impossible in hardware-based approaches due to the risk of cache overflow [21].

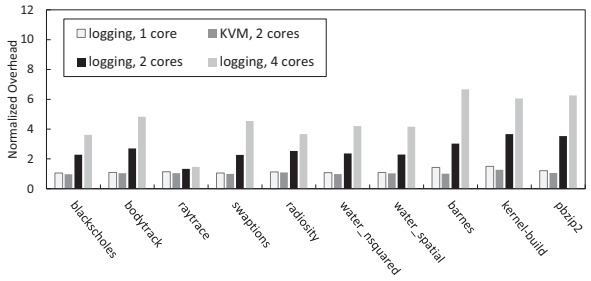


Figure 8: Recording overhead compared to the native execution.

5.4 Performance Overhead Compared to Native Execution

The performance overhead of a system can be evaluated in different ways. One way is to measure the overhead of the system relative to the base platform (e.g., KVM) it runs on. The problem with this approach is that the performance of different platforms can vary significantly and hence the overhead measured in this manner does not reflect the actual execution time of the system in real life. Consequently, we decide to compare the performance of our system to native execution, as shown in Figure 8.

As shown in the figure, the average performance overhead introduced by Samsara is 2.3× for recording computation intensive applications on two cores, and 4.1× on four cores. For I/O intensive applications, the overhead is 3.5× on two cores and 6.1× on four cores. This overhead is much smaller than the ones reported by prior works in software-only schemes, which cause about 16× or even 80× overhead when recording similar workloads on two or four cores [8, 27]. Samsara improves the recording performance dramatically because we avoid all memory access detections which are a major source of the overhead. Further experiment reveals that only 0.83% of the whole execution time is spent on handling page fault VM exits in Samsara, while prior CREW approaches suffer from more than 60% execution time spent on handling page fault VM exits.

Among the computation intensive workloads, barnes has a relatively high overhead (more than 3× on two cores), while retrace has a negligible overhead (about 0.3× on two cores). After analyzing the shared memory access pattern of these two workloads, we find that retrace contains many more read operations than write. Since Samsara does not trace any read accesses, these read operations do not cause any performance overhead. In contrast, barnes contains a lot of shared mem-

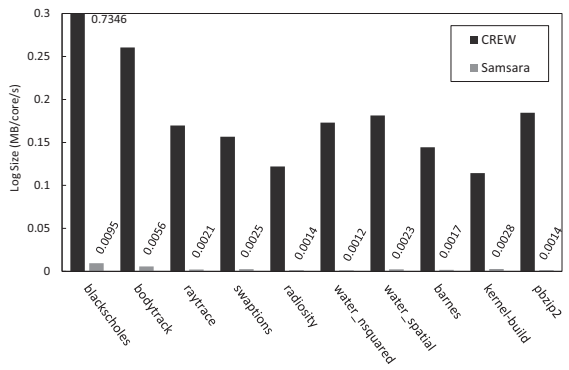


Figure 9: A comparison of the log file size between Samsara and CREW (4 cores, compressed with gzip).

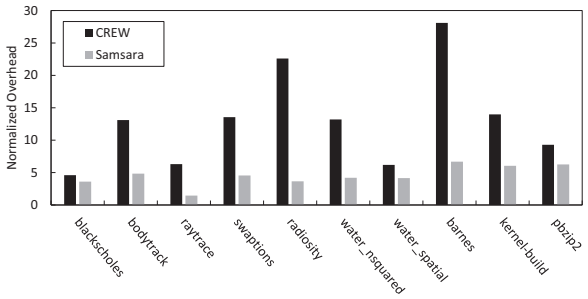


Figure 10: A comparison of recording overhead between Samsara and CREW (4 cores).

ory writes, and the unstructured communication pattern negates the effects of our hot page cache. Moreover, our page-level conflict detection may cause false conflicts (i.e., false sharing in SMP-ReVirt), which may lead to unnecessary rollback and increase performance overhead. When compared to computation intensive workloads, I/O intensive workloads incur relatively high overhead. This is also caused by the large number of concurrent I/O requests, which keep the chunk size quite small. Therefore, the execution time is not long enough to amortize the cost of the chunk commits in these workloads.

5.5 A Comparison with Prior software Approaches

To further evaluate our chunk-based strategy in Samsara against prior software-only approaches, we implement the original CREW protocol [13] in our testbed.

Log Size: Figure 9 shows the comparison against CREW protocol in log file size, in which Samsara reduces the log file size by 98.6% (i.e., from 0.22MB/core/s to 0.003MB/core/s). To understand the

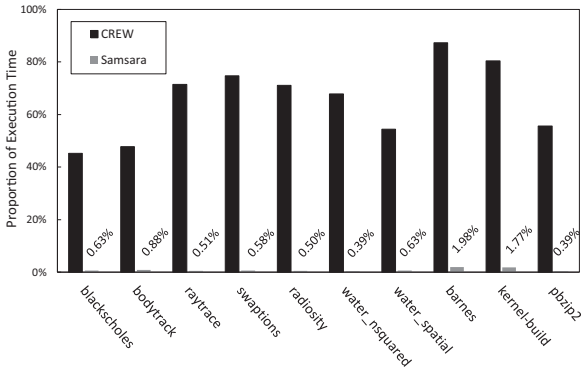


Figure 11: Proportion of the execution time consumed on handling page fault VM exits (4 cores).

improvement that Samsara achieves, we measure the proportions of each type of non-deterministic events in the log file. In this measurement, we find that nearly 99% of the events are memory access interleaving in CREW protocol, while only 10% of the events in Samsara are chunk commit orders.

Performance Overhead: We also compare the performance overhead of Samsara and the CREW protocol. The results in figure 10 illustrate that with four cores Samara reduces the overhead by up to 76.8% and the average performance improvement is 58.6% compared to the native execution.

Time Consumed on Handling Page Fault VM Exits: To understand why Samsara improves the recording performance so dramatically, we evaluate the time consumed on handling page fault VM exits in both approaches, since it is one of the primary contributors to the performance overhead. Figure 11 shows that 65.6% of the whole execution time is spent on handling page fault VM exits in the CREW protocol. In contrast, this proportion is only 0.83% in Samsara due to the HAV and chunk-based strategy we used.

5.6 Benefits of Performance Optimizations

Benefits of Caching Local Copies: Experimental results show that the average performance benefits contributed by caching local copies are 15.2% for recording computation intensive applications on four cores. For I/O intensive applications, the benefits increase to 22.1%. The effect of this optimization is highly dependent on the amount of temporal locality the local cache can exploit and the frequency of write operations. This explains why, for the applications, like water_nsquared and water_spatial, which perform few write operations and exhibit poor temporal locality, the benefits of this optimization are less (-1.24% and 2.76%).

Benefits of Double Buffering: The performance benefits contributed by double buffering are less significant. Empirically, the average performance increase is 4.61% when recording computation intensive applications on four cores. For I/O intensive applications, the improvement is 7.43%.

The improvement of the adaptive chunk size optimization is constrained by the frequent chunk truncations caused by I/O requests, thus is heavily application-specific. Among all the applications we experiment with, only a small subset of the computation intensive applications (e.g., raytrace from PARSEC, radiosity from SPLASH-2) is shown to have statistically significant benefit from this optimization.

6 Related Work

Deterministic Replay in Virtualization Environment: The idea of achieving deterministic replay based on virtualization environment was first proposed by Bressoud, et al. [6]. Similarly, ReVirt [12] can replay entire OSes deterministically and efficiently by recording all non-deterministic events within the VMM. ReTrace [40] is a trace collection tool based on the deterministic replay of the VMware hypervisor. However, both of them only work for uniprocessors and cannot be applied to multiprocessor environment. SMP-Revirt [13] is the first deterministic replay system that records and replays a multiprocessor VM on commodity hardware by leveraging CREW protocol. ReEmu [8] refines the CREW protocol with a seqlock-like design to achieve scalable deterministic replay in a parallel full-system emulator. While these virtualization-based schemes are flexible, extensible, and user-friendly, they suffer serious performance degradation and generate huge logs. In contrast, Samsara can leverage the latest HAV extensions in commodity multiprocessors to achieve efficient and practical deterministic replay. A preliminary description of this work was in [31].

Hardware-based Deterministic Replay: Hardware-based deterministic replay uses special hardware support for recording memory access interleaving. These schemes require modifications to the existing hardware, which increases the complexity of the circuits. FDR [38] records interleaving between pairs of instructions, and it improves the performance by implementing the Netzer's Transitive Reduction optimization [24] on hardware. RTR [39] extended FDR by only recording the logical time orders between memory access instructions. However, they still generate huge space overhead, which limits the duration of the recording. Strata [22] redesigns the recording strategy and records a stratum when a dependence occurs. Each stratum contains many memory operations issued by the corresponding processor since the last stratum is logged. Delorean [21] goes even further on this idea. Rather than logging individual dependence, it records memory access interleaving as series of chunks. By doing so, it allows out-of-order execution of instructions. IMMR [28] designs a chunk-based strategy for memory race recording in modern chip multiprocessors. Rerun [16] introduces an intermediate approach where it traces each data access but does not record this dependence. Instead, it records the number of instructions between two dependences. However, Rerun does not scale well during replay. To improve replay performance, Karma [3] is proposed as a chunk-based approach that aims to increase replay parallelism. Compared to chunk-based strategies in hardware schemes, Samsara improves the recording performance in VMM without requiring any hardware modification. Firstly, by leveraging HAV extensions, we avoid tracing every single memory access, instead perform a EPT traversal to obtain the read and write set. Secondly, we remove the centralized arbiter in Delorean, and propose a decentralized three-phase commit protocol to perform chunk commit efficiently.

7 Conclusion

In this paper, we have made the first attempt to leverage HAV extensions to achieve an efficient and practical software-based deterministic replay system on commodity multiprocessors. Unlike prior software schemes that trace every single memory access to record interleaving, we leverage the HAV extensions to track the read and write-set, and implement a chunk-based recording scheme in software. By doing so, we avoid all memory access detections, which are a major source of overhead in the prior work. In addition, we propose a decentralized three-phase commit protocol which significantly reduces the performance overhead by allowing chunk commits in parallel while still ensuring serializability. By evaluating our system on real systems, we demonstrate that Samsara can reduce the recording overhead from 10× to 2.3× and reduce the log file size to 1/70th on average.

Acknowledgments

The authors would like to thank Jon Howell, our shepherd Andreas Haeberlen, and the anonymous reviewers for their insightful comments. We also thank Yunqi Zhang for his valuable feedback on the earlier drafts of this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 61170056), the National Grand Fundamental Research 973 Program of China (Grant No. 2014CB340405).

References

[1] AGRAWAL, H., DE MILLO, R., AND SPAFFORD, E. An execution-backtracking approach to debugging. *Software, IEEE 8*, 3 (1991), 21–26.

[2] ALTEKAR, G., AND STOICA, I. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), pp. 193–206.

[3] BASU, A., BOBBA, J., AND HILL, M. D. Karma: Scalable deterministic record-replay. In *Proceedings of the International Conference on Supercomputing* (2011), ICS '11, pp. 359–368.

[4] BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIĆ, M., MIHOČKA, D., AND CHAU, J. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (2006), VEE '06, pp. 154–163.

[5] BIENIA, C. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (1995), SOSP '95, pp. 1–11.

[7] CHEN, A., MOORE, W. B., XIAO, H., HAEBERLEN, A., PHAN, L. T. X., SHERR, M., AND ZHOU, W. Detecting covert timing channels with time-deterministic replay. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), pp. 541–554.

[8] CHEN, Y., AND CHEN, H. Scalable deterministic replay in a parallel full-system emulator. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2013), pp. 207–218.

[9] CHEN, Y., HU, W., CHEN, T., AND WU, R. Lreplay: A pending period based deterministic replay scheme. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, pp. 187–197.

[10] DAH-MING, C., AND RAJ, J. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems 17*, 1 (1989), 1–14.

[11] DEVECSERY, D., CHOW, M., DOU, X., FLINN, J., AND CHEN, P. M. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), pp. 525–540.

[12] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation* (2002), pp. 211–224.

[13] DUNLAP, G. W., LUCCHETTI, D. G., FETTERMAN, M. A., AND CHEN, P. M. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2008), pp. 121–130.

[14] HONARMAND, N., DAUTENHAHN, N., TORRELLAS, J., KING, S. T., POKAM, G., AND PEREIRA, C. Cyrus: Unintrusive application-level record-replay for replay parallelism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13, pp. 193–206.

[15] HONARMAND, N., AND TORRELLAS, J. Relaxreplay: Record and replay for relaxed-consistency multiprocessors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), pp. 223–238.

[16] HOWER, D. R., AND HILL, M. D. Rerun: Exploiting episodes for lightweight memory race recording. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008), ISCA '08, pp. 265–276.

[17] JOSHI, A., KING, S. T., DUNLAP, G. W., AND CHEN, P. M. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (2005), SOSP '05, pp. 91–104.

[18] KING, S. T., AND CHEN, P. M. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003), SOSP '03, pp. 223–236.

[19] LEBLANC, T., AND MELLOR-CRUMMEY, J. Debugging parallel programs with instant replay. *Computers, IEEE Transactions on C-36*, 4 (April 1987), 471–482.

[20] LEE, D., WESTER, B., VEERARAGHAVAN, K., NARAYANASAMY, S., CHEN, P. M., AND FLINN, J. Respec: Efficient online multiprocessor replayvia speculation and external determinism. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems* (2010), pp. 77–90.

[21] MONTESINOS, P., CEZE, L., AND TORRELLAS, J. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture* (2008), pp. 289–300.

[22] NARAYANASAMY, S., PEREIRA, C., AND CALDER, B. Recording shared memory dependencies using strata. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), pp. 229–240.

[23] NARAYANASAMY, S., POKAM, G., AND CALDER, B. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32Nd Annual International Symposium on Computer Architecture* (2005), ISCA '05, pp. 284–295.

[24] NETZER, R. H., AND XU, J. Adaptive message logging for incremental program replay. *IEEE Concurrency 1*, 4 (1993), 32–39.

[25] OLSZEWSKI, M., ANSEL, J., AND AMARASINGHE, S. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 97–108.

[26] PARK, S., ZHOU, Y., XIONG, W., YIN, Z., KAUSHIK, R., LEE, K. H., AND LU, S. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), pp. 177–192.

[27] PATIL, H., PEREIRA, C., STALLCUP, M., LUECK, G., AND COWNIE, J. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (2010), pp. 2–11.

[28] POKAM, G., PEREIRA, C., DANNE, K., KASSA, R., AND ADL-TABATABAI, A.-R. Architecting a chunk-based memory race recorder in modern cmps. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture* (2009), MICRO 42, pp. 576–585.

[29] QIAN, X., HUANG, H., SAHELICES, B., AND QIAN, D. Rainbow: Efficient memory dependence recording with high replay parallelism for relaxed memory model. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on* (Feb 2013), pp. 554–565.

[30] QIAN, X., SAHELICES, B., AND QIAN, D. Pacifier: Record and replay for relaxed-consistency multiprocessors with distributed directory protocol. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014), ISCA '14, pp. 433–444.

[31] REN, S., LI, C., TAN, L., AND XIAO, Z. Samsara: Efficient deterministic replay with hardware virtualization extensions. In *Proceedings of the 6th Asia-Pacific Workshop on Systems* (2015), APSys '15, pp. 9:1–9:7.

[32] SCALES, D. J., NELSON, M., AND VENKITACHALAM, G. The design of a practical system for fault-tolerant virtual machines. *SIGOPS Oper. Syst. Rev. 44*, 4 (Dec. 2010), 30–39.

[33] SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference, General Track* (2004), pp. 29–44.

[34] VEERARAGHAVAN, K., LEE, D., WESTER, B., OUYANG, J., CHEN, P. M., FLINN, J., AND NARAYANASAMY, S. Doubleplay: Parallelizing sequential logging and replay. *ACM Trans. Comput. Syst. 30*, 1 (Feb. 2012), 3:1–3:24.

[35] VOSKUILEN, G., AHMAD, F., AND VIJAYKUMAR, T. N. Timetraveler: Exploiting acyclic races for optimizing memory race recording. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA '10, pp. 198–209.

[36] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture* (1995), ISCA '95, pp. 24–36.

[37] WU, X., AND MUELLER, F. Elastic and scalable tracing and accurate replay of non-deterministic events. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (2013), ICS '13, pp. 59–68.

[38] XU, M., BODIK, R., AND HILL, M. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture* (2003), pp. 122–133.

[39] XU, M., HILL, M. D., AND BODIK, R. A regulated transitive reduction (rtr) for longer memory race recording. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), ASPLOS XII, pp. 49–60.

[40] XU, M., MALYUGIN, V., SHELDON, J., VENKITACHALAM, G., AND WEISSMAN, B. Retrace: Collecting execution trace with virtual machine deterministic replay. In *Proceedings of the Third Annual Workshop on Modeling, Benchmarking and Simulation* (2007).

[41] YANG, Z., YANG, M., XU, L., CHEN, H., AND ZANG, B. Order: Object centric deterministic replay for java. In *USENIX Annual Technical Conference* (2011).

[42] ZHU, J., JIANG, Z., AND XIAO, Z. Twinkle: A fast resource provisioning mechanism for internet services. In *Proceedings of the IEEE INFOCOM* (2011), pp. 802–810.

[43] ZHU, J., JIANG, Z., XIAO, Z., AND LI, X. Optimizing the performance of virtual machine synchronization for fault tolerance. *IEEE Transactions on Computers 60*, 12 (Dec 2011), 1718–1729.

Hardware-Assisted On-Demand Hypervisor Activation for Efficient Security Critical Code Execution on Mobile Devices

Yeongpil Cho[1] , Junbum Shin[2] , Donghyun Kwon[1]
MyungJoo Ham[2] , Yuna Kim[2] , Yunheung Paek[1]
[1]*Seoul National University* [2]*Samsung Electronics*

Abstract

As more and more mobile applications need to run security critical codes (SCCs) for secure transactions and critical information handling, the demand for a Trusted Execution Environment (TEE) to ensure safe execution of SCCs is rapidly escalating. Although a number of studies have implemented TEEs using TrustZone or hypervisors and have evinced the effectiveness in terms of security, they face major challenges when considering deployment in mobile devices. TrustZone-based approaches bloat the TCB of the system as they must increase the code base size of the most privileged software. Hypervisor-based approaches incur performance overhead on mobile devices that are already suffering from resource restrictions.

To alleviate these problems, in this paper, we propose a hybrid approach that utilizes both TrustZone and a hypervisor. Our approach basically implements a TEE using a hypervisor, while mitigating performance overhead by activating the hypervisor only when the TEE is demanded by SCCs. This scheme, called on-demand hypervisor activation, has been efficiently and securely implemented by leveraging the memory protection capability of TrustZone. We have implemented and experimented our system with real world applications. The results show that our system can successfully protect SCCs without any noticeable delay ($< 100\ \mu$s), while limiting the overhead increase due to our hypervisor during its hibernation near 0 %.

1 Introduction

With a plethora of mobile devices, an extensive range of mobile applications providing convenience are emerging into our lives. However, as mobile devices increasingly offer more sophisticated services, security and sensitivity of the data they handle has become a critical issue [31]. Mobile payment applications nowadays, for example, enable customers to purchase diverse products regardless of place or time. For this, the applications must be authorized to process sensitive data, such as credit card information and personal identification numbers. Accordingly, a number of recent attacks on mobile devices for monetary gain have mostly aimed at achieving unlawful access to sensitive data in such applications. To fend off these attacks, many engineers have introduced the notion of *privilege separation* in the development of their applications by utilizing trusted execution environments (TEEs) on mobile devices. The key objective of privilege separation is to minimize the attack surface of sensitive data by limiting the data accessibility only to the trusted parts of an application, called *security critical code* (SCC) [36, 37, 33], that will be partitioned away from the rest of the application at code development time, and deployed exclusively for secure data transactions in the TEE at runtime.

To provide TEEs for privilege separation on mobile devices, a growing amount of work [23, 48, 45, 44] leverages TrustZone [2], which is a hardware-based security extension installed in ARM processors. TrustZone maintains two separate execution environments, the normal world and the secure world. The normal world is reserved for common OSes and untrusted applications. These typically have rich functionality but are prone to potential attacks due to the existence of exploitable vulnerabilities [13, 61]. Whereas in the secure world, a trusted minimal OS is installed to establish a TEE and provide an individual secure execution environment for each SCC at runtime. Unfortunately, this approach, relying on TrustZone to protect SCCs, faces a major challenge in terms of security. In TrustZone, as the secure world is undoubtedly the trusted computing base (TCB) of the entire system due to its highest privilege level, it must maintain integrity to ensure the safety of the system. However, to support the growing number of versatile SCCs, functional extensions of the trusted OS are inevitable, which increases the size and complexity of the

code base of the trusted OS. Recall that, even carefully designed and engineered code contains bugs and vulnerabilities in proportion to its size [38]. Hence, hosting more SCCs in the secure world may open more doors for attackers to compromise the trusted OS by exploiting its vulnerabilities, which in turn may jeopardize the safety not only of the secure world but also of the entire system. In the real world, as a result, mobile device vendors, such as Samsung, usually have a tendency of being reluctant to render the TrustZone-based secure world freely accessible to public developers. Instead, they have only accepted a few OEM applications that have passed thorough in-house testing.

An alternate way to provide a TEE would be to use a hypervisor which, as the most privileged software layer, is responsible for monitoring and controlling the behavior of the OS layer below it. As long as it is carefully designed, the hypervisor can provide these security and isolation guarantees even when the OS is compromised. Therefore, several studies have relied on hypervisors to implement a TEE for SCC protection and have demonstrated the safety and feasibility of their approaches [36, 62]. However, it must be noted [12, 40, 43] that a hypervisor, running as an extra software layer for virtualization in the system, inevitably suffers from non-negligible performance degradation. This performance overhead may particulary be of great concern in mobile devices which are mostly restricted by severe resource constraints. In fact, the performance concern has been considered to be one of the primary reasons that impedes a wide adoption of existing hypervisor-based approaches in such small resource-stringent devices.

Based on our observations on the problems of previous approaches using either TrustZone or a hypervisor, we have developed a new hybrid approach that attempts to take advantages of both a hypervisor and TrustZone in a way to attain safe, yet efficient SCC execution on mobile devices. To limit the extension of the secure world in TrustZone-based approaches, our approach uses a hypervisor to implement an additional TEE in the normal world alongside the original TEE in the secure world. This *virtualization* scheme enables application developers to implement and distribute their SCCs without the security concern for the secure world corresponding to the system TCB. To tackle the performance concern of other hypervisor-based approaches, we have devised a scheme, called *on-demand hypervisor activation*, which activates our hypervisor only when a TEE must be established for SCC executions. In reality, SCCs are executed occasionally just by a handful of special security applications installed in the system such as DRM and certificate managements. Also an earlier study [14] revealed that even for a given security application, SCC often accounts for a small portion of the entire application. All these support our assertion that our hypervisor should be deactivated for most of the time while the system is up and running. Therefore, as being compared to other approaches which maintain their hypervisors persistently at all times, our solution will suffer from much less virtualization overhead.

To confirm the feasibility of our hybrid approach, we have designed a protection system, named *On-demand Software Protection* (OSP). OSP relies on a hypervisor to meet security requirements for ensuring safe executions of SCCs by implementing an additional TEE in the normal world while suppressing the TCB bloating of the secure world. Therefore, mobile device vendors can allow public developers to install and exeucte their SCCs in the TEE without a large amount of verification efforts. OSP also meets the stringent performance requirements of mobile devices by adopting an on-demand hypervisor activation scheme. In our design, we use TrustZone to enforce memory protection when our hypervisor is deactivated. While the hypervisor is active and running on the machine, OSP checks if there are any SCCs currently being executed by an application. As soon as it finds that no SCC is running, it deactivates its hypervisor and simultaneously orders TrustZone to protect the current states of the deactivated hypervisor as well as all the SCCs that were protected by the hypervisor. TrustZone internally maintains a secure enclave that is not accessible to any other software including the OS kernel. Therefore, every critical information about the hypervisor and SCCs will be safely protected while the hypervisor is in hibernation. Later when an application is about to invoke one of the SCCs, OSP removes the protection of TrustZone, and wakes up the hypervisor by reactivating it with its original states that were protected by TrustZone. Then the activated hypervisor soon reconstructs the TEE where the newly invoked SCC will be securely executed.

To evaluate OSP, we have implemented a prototype of OSP on a development board for Exynos 5422, an ARM-based application processor (AP) platform adopted by commercial mobile devices like Samsung Galaxy S5. In implementation of the OSP prototype, we have only utilized the existing hardware features available in most ARM APs; thus, we believe that OSP is deployable on COTS-devices as well. In order to evaluate its feasibility, we have ported some Android applications to OSP: e.g., Chromium web browser and a file encryption application. The results revealed that OSP was able to ensure secure executions of all SCCs in our system.

The rest of this paper is structured as follows. Section 2 provides background information. Section 3 discusses our threat model and assumptions. Section 4 describes the design and Section 5 introduces implementation details of the OSP prototype. Section 6 presents the evaluation of the experimental results and Section 7 discusses

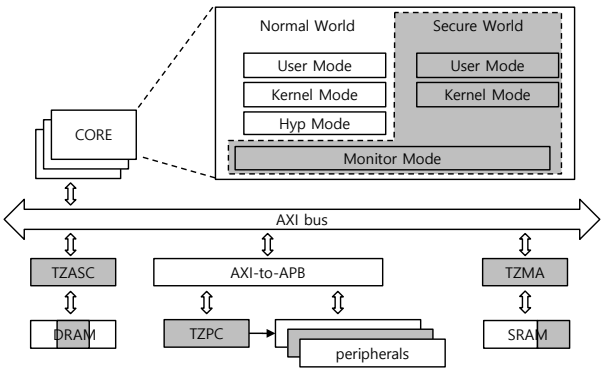


Figure 1: Components of ARM TrustZone

remaining issues. Finally, Section 8 shows related works and Section 9 concludes this paper.

2 Background

This section provides a summary of the security and virtualization extensions supported by ARM.

2.1 Security extensions

TrustZone enables the system to operate in both the secure and normal world in a time-sliced fashion. To separate the two worlds and ensure the confidentiality and integrity of the secure world, diverse extensions are integrated across the system, as depicted in Figure 1. First, the secure and normal world have their own processor modes and system configuration registers and are therefore allowed to build individual software stacks for the OS and applications even if they share a single physical system. To coordinate and arbitrate between the two worlds, the most privileged processor mode, called the monitor mode, is added alongside the existing processor modes. Both the secure and the normal world are able to enter the monitor mode by issuing a secure monitor call (SMC) instruction.

TrustZone includes an extension for secure interrupts as well, which is only visible and delivered to the secure world. The ARM architecture is equipped with a GIC [8] to control system-wide interrupts in a manner similar to APIC of Intel. GIC provides 16 software generated interrupts (SGI) that can be delivered to every core or only to specific cores as inter-processor interrupts (IPI). The security extension of GIC allows us to designate some of the SGIs as secure interrupts such that they can be used to pass signals secretly. GIC also enables secure SGIs using FIQ signals, instead of IRQ signals, in order to increase the priority of the interrupts.

The NS-bit in the secure configuration register (SCR) indicates whether a processor is executing in the normal world or the secure world. This bit is also propagated across the entire system by being attached to system bus transactions, so that scattered TrustZone components are able to manage access to resources, such as memory and peripherals, out of the CPU cores. For example, TrustZone includes TZMA [4] and TZASC [7], respectively located in front of the SRAM and DRAM. They partition the address spaces corresponding to SRAM and DRAM into several regions, each of which is assigned to the secure world or the normal world, and prevent access to the secure world regions from the normal world. TrustZone also adds the TZPC [5], which enforces a similar security policy with regard to peripherals. This way, the secure world can configure and access peripherals in an explicit manner.

2.2 Virtualization extensions

Similar to VT-x [39] of Intel and SVM [29] of AMD, ARM introduced hardware virtualization extensions [6] that allow hypervisors to efficiently manage guest OSes. To empower hypervisors to configure the entire system, ARM supports a privileged processor mode known as the hyp mode [6], which is beneath the kernel mode in the hierarchy of processor modes as described in Figure 1. Hypervisors running in the hyp mode are able to configure fundamental system resources, such as the exception vector table, counter and timer, with a variety of control registers only accessible in the hyp mode. In particular, hypervisors can configure and deploy the extended page tables underneath the primary page tables managed by guests. By assigning various access-permission flags in the extended page tables, hypervisors are able to exclusively enforce access-control policies for all address spaces of guests. Along with the hyp mode, a hypervisor call (HVC) instruction is added for communication between hypervisors and guests.

The ARM virtualization extensions include a system MMU [3] as well. If the system MMU is enabled, each peripheral is given its own page table. Configuring those page tables according to guests, hypervisors can dynamically change the address spaces of peripherals. This facilitates device virtualization without the intervention of hypervisors, thereby improving hypervisors in terms of their performance and porting effort. The system MMU, moreover, is effective at preventing DMA attacks of misconfigured peripherals by limiting the accessible address space of each peripheral.

3 Threat model and Assumptions

In this section, we describe the threat model and assumptions pertaining to the implementation and design of OSP.

Threat model. We assume that our adversaries can exploit vulnerabilities to gain full control over the rich OS. In other words, they can freely perform arbitrary memory reads, memory writes, and code executions in the address spaces of the OS kernel and applications. With this capability, they may attempt to access the address spaces of SCCs in order to steal confidential contents revealed during runtime. They may also try to acquire the binary files of SCCs so that they could extract statically stored secrets with reverse-engineering or determine the core algorithms of SCCs through binary analysis.

In addition, as we cannot fully trust application developers and their products, SCCs could be abused to tamper with and/or eavesdrop on other SCCs and their sensitive data. Malicious SCCs may also attempt to subvert a TEE by making arbitrary system calls with crafted parameters.

Assumptions. We assume that in the secure world, carefully verified software is preinstalled and dynamic software installation is not allowed. The built-in software of the secure world, including the minimal OS and OEM applications, is trusted and will be intactly loaded with a secure boot mechanism such as AEGIS [50] or UEFI [56]. Therefore, we do not take into account any attacks originating within the secure world. We also do not consider denial-of-service (DoS) attacks. Memory attacks, such as cold boot attacks [26] and bus monitoring attacks [49, 54] are beyond the scope of our adversary model as well. Similarly, hardware attacks, such as physical side-channel and JTAG attacks are not considered in this work.

4 Design

OSP creates a TEE alongside Trustzone, which provides a security and efficient protection mechanism. This TEE can be used by mobile device vendors to provide a way for application developers to protect their SCCs. In this section, we present the details of the design of OSP and explain how it achieves this goal.

4.1 Design objectives

In order to secure SCCs, we deliberately design OSP while seeking to accomplish the following objectives.

Practical mechanism. Opening the secure world for SCC protection causes a security concern about TCB bloating from an increased code base. Therefore, OSP should arrange a TEE on the exterior of the secure world, thereby enabling application developers to protect their SCCs without reducing the level of security of the secure world. In addition, as we consider resource constrained mobile devices, OSP should incur negligible performance overhead when maintaining the TEE.

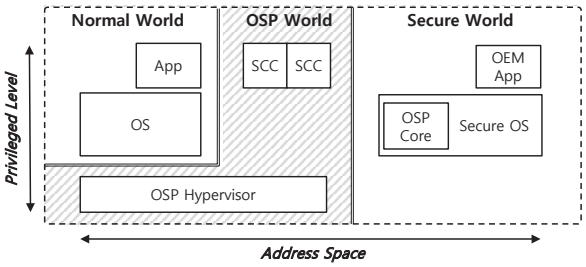


Figure 2: Overview of OSP. OSP consists of the OSP hypervisor, which protects and manages the SCCs, and the OSP core, which controls and configures OSP overall.

Runtime protection. To protect the confidentiality and integrity of SCCs, OSP should provide each SCC an individual execution environment, which is isolated from the OS kernel and other SCCs. As SCCs are not trusted entities, each SCC should be able to communicate with other SCCs and the OS kernel only when allowed by OSP.

Secure provisioning. SCC binaries often encompass secrets, such as key values and core algorithms, which developers want to protect. Therefore, SCC binaries should not be exposed to attackers during their distribution so as to ensure the confidentiality of those secrets.

4.2 Overall Design

Figure 2 depicts the overall design of OSP. OSP defines the OSP world alongside the normal world and the secure world. As the OSP world is completely separated from both worlds, OSP can securely provide an additional TEE to SCCs while keeping the secure world compact. OSP consists of two software components: the OSP core and the OSP hypervisor. As the TCB of the entire OSP system, the OSP core, located in the secure world, is responsible for initializing OSP during the system boot sequence and for deploying and controlling the OSP hypervisor at runtime. The OSP hypervisor, the de facto TCB of SCCs, plays a vital role in the functionality of OSP. It protects the OSP world by blocking unauthorized accesses of the normal world; it also creates a TEE in the OSP world, thereby providing isolated execution environments for SCCs.

Although the OSP hypervisor is a fundamental component in OSP for the runtime protection of SCCs, it may incur non-negligible performance impacts due to virtualization overheads. Therefore, to minimize such overheads, OSP activates its hypervisor only while a protection service is required, i.e., when one or more SCCs are running. Moreover, the OSP core expands the secure world enough to cover the entire OSP world to protect it from invasions by the normal world when the OSP hypervisor is no longer active.

Function Name	Parameter	Call-site	Description
Management interfaces			
SCC_register	scc_file_name, ptr_external_handler	app	Registers an SCC with a specification. Upon success, returns the SCC's number.
SCC_unregister	scc_num	app	Unregisters an SCC.
SCC_parameter_add	ptr_scc_param_spec, param_flag, ptr_param, length	app	Add a parameter to a parameter specification.
SCC_invoke	scc_num, entry_func, ptr_scc_param_spec, arg0...arg3	app	Invokes an SCC with a parameter specification. Upon finish, returns a return value.
SCC_ret_to_scc	scc_num, return_value	app	Return to an SCC with a return value
Service interfaces			
OSP_save	ptr_data, length	SCC	Save data on secure storage. Upon success, returns the storage number.
OSP_load	storage_num, ptr_buffer, length	SCC	Loads the data for a storage number.
OSP_delete	storage_num	SCC	Deletes the data for a storage number.
OSP_encrypt	ptr_data, ptr_buffer, length	SCC	Encrypt data
OSP_decrypt	ptr_data, ptr_buffer, length	SCC	Decrypt data
OSP_signing	ptr_data, length, private_key, signature	SCC	Sign data with a given private key
OSP_verification	ptr_data, length, public_key, signature	SCC	Verify data with a given public key
OSP_external_handler	cmd, arg0...arg3	SCC	Call the external handler with parameters. Upon finish, returns a return value

Table 1: The management and service interfaces of OSP

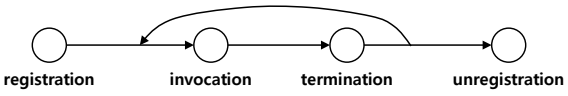


Figure 3: The lifecycle model of an SCC

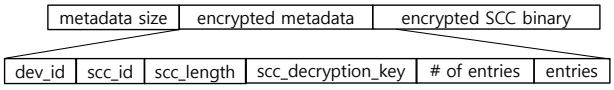


Figure 4: The format of an SCC file

4.3 Development of SCCs

To protect sensitive data using SCCs, developers need to develop their applications while being conscious of the concept of privilege separation. Developers should handle sensitive data only in SCCs and should transmit the data to the remainder of their applications after encrypting it to prevent exposure. For the sake of minimizing the attack surface, we highly recommend that developers ensure that their SCCs are self-contained to prevent internal states from being exposed outside of the SCCs during execution. However, SCCs may sometimes want to outsource certain functions, such as network or file system access, to enrich their functionality. OSP supports such cases by letting developers implement external handlers that can process outsourced requests in their applications running on the rich OS and allowing SCCs to call those handlers.

Application developers should design SCCs considering the lifecycle model of an SCC, depicted in Figure 3. They can implement SCCs using the following interfaces that are offered in the form of a static or dynamic library. We describe the details in Table 1.

Management interface. Using the management interface, a developer can include an SCC into her application as if using a dynamic library. To begin with, we assume that there is a prebuilt SCC file (§4.4). The developer should initially call `SCC_register` with the name of the SCC file and, if needed, the address of an external handler located in the application. An SCC number is then given after registration, which is used to specify an SCC in the later invocation and unregistration processes. To invoke the registered SCC, the developer should prepare a parameter specification by gathering the properties of the parameters that are to be passed, each of which consists of a start address, a length and flags. In particular, the flags specify when a parameter will be marshalled; the `input` and the `output` flags indicate that the corresponding parameter will be marshalled when the SCC is invoked and returned, respectively, and the `shared` flag means that the corresponding parameters do not need to be marshalled because they are shared between the application and the SCC. At this point, the developer can invoke the SCC with the specified parameters and can continue to invoke it unless the SCC is unregistered.

Service interface. The current implementation of OSP provides secure storage and cryptographic services, which allows SCCs to protect passwords and cryptographic keys. Expanding the capabilities of SCCs by adding new services offered by the OSP is left for future work (refer to Section 7). In the current OSP, instead, SCCs can outsource some operations that are not supported by OSP, i.e., memory management, networking, file system, to an external handler, which would be appointed during the SCC registration step, located in the application. However, as external handlers may be po-

tentially vulnerable, developers are responsible for verifying returned results to defeat unintended attacks such as Iago attacks [16].

4.4 Provision of SCCs

SCCs can be distributed in various ways. In this paper, we assume a scenario where application developers distribute their SCCs along with their applications. Application developers initially need to have a developer ID (e.g., a developers private key) that can identify them individually. Because developers are allowed to distribute more than two SCCs, they should choose a unique SCC ID for distinguishing each SCC. To maintain integrity and confidentiality, SCCs must be distributed in an encrypted form. Figure 4 describes the distribution file format of an SCC, which is made up of metadata and an encrypted SCC binary. The metadata consists of two noted IDs and a key for decrypting the encrypted SCC binary. In addition, the metadata should contain a list of entry functions that are allowed for applications to invoke; thus preventing non-designated internal functions from being called directly. Lastly, the metadata is encrypted asymmetrically with the public key of OSP to protect the contents by sealing.

4.5 Execution of SCCs

At runtime, once the OSP hypervisor receives a registration request with an encrypted SCC file, the hypervisor copies contents of the file to the OSP world and performs a series of decryptions and parsing. It first decrypts the metadata of the SCC with the private key of OSP and parses that to extract developer and SCC IDs and the decryption key. The hypervisor subsequently decrypts the encrypted SCC binary and begins to load the decrypted contents onto the OSP world. It prepares an empty extended page table and maps the address space of code, data and stack of the SCC to the page table. Then, it finalizes the registration step by returning the number of the SCC to the caller application.

An invocation request for the SCC is delivered to the OSP hypervisor with a parameter specification and an entry functions number. In the beginning, parameters that are documented on the parameter specification are marshalled according to the details of the specification. Next, the hypervisor maps the parameters to the page table of the SCC. The hypervisor masks unrelated interrupts to the SCC, preventing the OS kernel from interrupting the execution of the SCC. The hypervisor, moreover, applies the prepared page table to the system to reflect the SCCs own address space. Finally, it checks the correctness of the passed entry functions number and it transfers control to the corresponding entry function. At the same time, the hypervisor instigates a timeout to prevent a buggy or malicious SCC from seizing the system for too long. After a while, if the SCC finishes its work, the OSP hypervisor instantly restores the states relevant to the caller application and returns the control to the application. A similar procedure is conducted when the SCC calls its external handler while it is being executed. In this case, however, it is required for the application to issue `OSP_ret_to_scc` in order to resume the SCC.

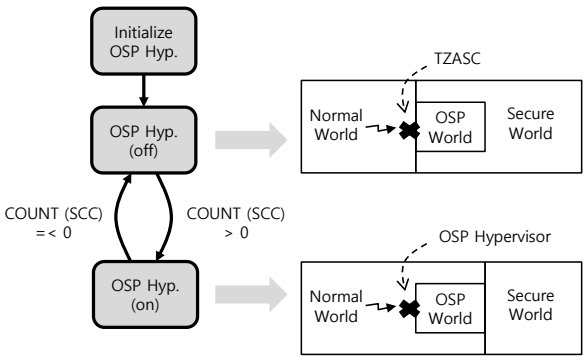


Figure 5: Two protection mechanisms for the OSP world

The execution environment of the SCC is maintained until the application unregisters the SCC explicitly. If an unregistration request is issued, the OSP hypervisor completely clears every relevant state of the SCC, such as the page table and the contents of the heap and stack regions. To use the same SCC from that time, the application must register it again.

4.6 On-demand activation of the OSP hypervisor

As noted above, the OSP hypervisor is activated by the OSP core only while SCCs are running in the OSP world. The OSP core finishes the default configuration of the OSP hypervisor during the boot sequence. This process includes the creation of the default extended page table that identically maps the entire address space of the normal world, although the OSP core does not enable extended paging at this point. However, considering that the OSP hypervisor depends on extended paging, to protect the OSP world while the hypervisor is deactivated, introducing another mechanism is inevitable.

For this purpose, OSP capitalizes on TZASC which, as a hardware component of TrustZone, allows dynamically setting the address space of the secure world. While the extended paging is disabled, as described in Figure 5, the OSP core includes the OSP world in the secure world using TZASC, thereby preventing malicious accesses originating from normal world software. Note that OSP creates its TEE in the normal world rather than

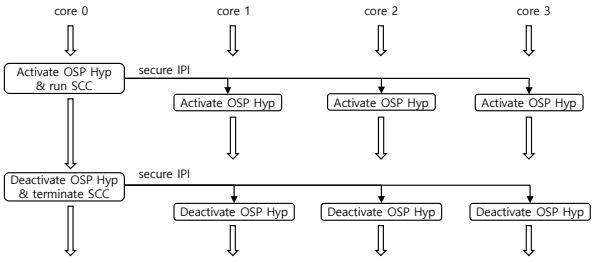


Figure 6: On-demand activation of the OSP hypervisor in multi-core environments

```
Procedure ACTIVATE_OSP_HYP
  Enable the extended paging
  Send secure IPIs to other cores to enable the extended paging, too
  Reduce the secure world to reveal the OSP world using TZASC
End

Procedure DEACTIVATE_OSP_HYP
  Expand the secure world to cover the OSP world using TZASC
  Clean and invalidate cache entries of the OSP world
  Disable the extended paging
  Send secure IPIs to other cores to disable the extended paging, too
End
```

Figure 7: On-demand activation and deactivation routines of the OSP core

the secure world. Therefore, this configuration is cleared when SCCs are invoked; from this point on, the OSP hypervisor protects the OSP world from the normal world by activating extended paging. When an SCC is terminated, the OSP hypervisor checks if any other SCC is still running. When all SCCs are terminated, the OSP core disables extended paging to reduce the performance degradation caused by extended paging. This, however, renders security-critical data stored in the OSP world vulnerable to untrusted software in the normal world because they are now accessible to anyone with control over the normal world. To address this problem, before disabling extended paging, the OSP core reconfigures TZASC so that the secure world once again engulfs the OSP world.

4.6.1 Multi-core support

OSP supports multi-core environments; it allows several SCCs to run concurrently in different cores. However, this does not mean that the OSP hypervisor can be individually activated in each core, even though each core has its own MMU and control registers. Because there is only one TZASC within the system, located between the system bus and main memory, when a core configures TZASC, the effect is not limited to that core. For example, when in a core, if the OSP core configures TZASC to pull the OSP world from the secure world and activates the OSP hypervisor, the OSP world will immediately be exposed to normal world software on all of the other cores. In addition, another severe problem will arise when the OSP hypervisor is deactivated. Let us assume that, in a core, the OSP core deactivates the OSP hypervisor and reconfigures TZASC to include the OSP world in the secure world. At this point, however, the OSP hypervisor is still activated in the other cores, a permission violation for the secure world will be provoked (at least due to address translations by extended paging). Consequently, OSP must synchronize the hypervisor activation state of every core, as in Figure 6.

4.6.2 Activation and deactivation routines

Figure 7 summarizes the routines of the OSP core for activating and deactivating the OSP hypervisor at runtime. If the activation routine is initiated, the routine initially enables extended paging and sends secure IPIs to other cores, so that they enable extended paging as well, to activate the OSP hypervisor. This must be done before removing the protection of TZASC in order to prevent untrusted software from accessing the OSP world. We can control extended paging using the Hyp Configuration Register (HCR), which is not accessible in the normal world. The HCR register consists of a number of configuration bits; in particular, we can enable and disable extended paging by setting and clearing the VM-bit. In addition, the HCR register contains the TDC-bit. This bit makes OSP enable cache memory while SCCs run even if address translation of the kernel space is disabled. After activating the OSP hypervisor, the routine controls TZASC to reveal the OSP world from the secure world. TZASC can be controlled using memory mapped registers similar to most components of ARM. As explained in Section 2.1, TZASC manages regions as a unit of permission enforcement. We can control these regions with two primary registers, the Region Setup Register and Region Attributes Register. The former one controls the base address of each region. The latter one plays a more important role; it determines the size and permission[1]. Particularly, this register has an enable bit, so that we can enable and disable the corresponding region by toggling the bit.

The deactivation routine is performed in the reverse order of the activation routine. First, it configures TZASC to cover the OSP world with the secure world. After configuring TZASC, the routine cleans and invalidates every cache entry corresponding to the OSP world. Otherwise, some states of the OSP world remaining in the cache memory may be exposed to untrusted software in the normal world. Finally, the routine deactivates the OSP hy-

[1] A permission can be configured as a combination of secure read, secure write, non-secure read and non-secure write flags.

pervisor by disabling extended paging. As the last step, it sends secure IPIs to the other cores so that they can also disable extended paging. Each core disables extended paging as soon as it receives the IPI.

4.7 Interface implementation

As noted in Section 4.3, OSP provides two types of interfaces, called the management interface and the service interface. In this subsection, we explain how OSP implements these interfaces.

Management interface. In general terms, the normal world software is intended to communicate with a hypervisor using the HVC instruction. However, this option is not available in OSP considering the dynamic activation state of the OSP hypervisor. Thus, OSP would have to provide two duplicate management interfaces which are implemented based on the SMC and the HVC instructions, and the normal world software would need to choose the proper interface each time depending on whether or not the OSP hypervisor is hibernating in the secure world.

To avoid this complication, OSP implements the management interface using only the SMC instruction. For this, the activation routine of the OSP core, introduced in Section 4.6.2, sets the TSC-bit of the HCR register to 1, thus trapping future executions of the SMC instruction into the OSP hypervisor. After which, if a normal world software executes SMC instructions, the OSP hypervisor first analyzes whether the SMC instruction is intended for the interface of OSP by parsing it. If so, the hypervisor performs a management operation according to the request, and if not, it is transferred to the OSP core to be handled in the secure world.

Service interface. It is fairly known that the supervisor call (SVC) instruction is used to implement system calls of the kernel. Moreover, (unprivileged) applications are not allowed to execute the SMC or the HVC instructions on ARM. Accordingly, OSP enables SCCs to use the service interface that is implemented based on the SVC instruction. For this, the activation routine of the OSP core sets the TGE-bit of the HCR register. By doing so, all executions of the SVC instructions are trapped into the OSP hypervisor; thus, the hypervisor can receive and handle service requests of SCCs.

5 Implementation

In this section, we explain implementation details which were not presented in the earlier sections.

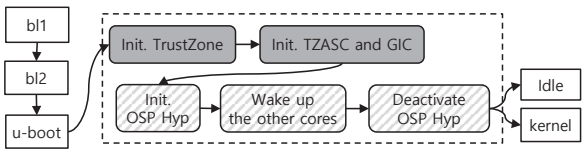


Figure 8: Boot sequence of OSP

5.1 OSP Hypervisor

The structure of our OSP hypervisor is somewhat related to KVM/ARM [21], an open-source hypervisor found in the mainline Linux kernel, in the sense that our hypervisor borrows several key implementation mechanisms regarding virtualization. However, in comparison to KVM/ARM, the OSP hypervisor has a simple structure with a small code base because it only needs to support a single guest OS. As a result, it can run with much lower overhead than the general-purpose KVM hypervisors. For example, the OSP hypervisor requires only a quarter to half of the CPU cycles (1,119 cycles) required by KVM/ARM (from 2,112 to 4,917 cycles) for world-switching latency (round trip from the kernel to the hypervisor).

For simplicity, we statically place the OSP hypervisor on the top 128 MB of the physical memory address that is reserved for the OSP world. Such static deployment may reduce the available physical memory of the kernel, but this problem could be mitigated by making the OSP hypervisor use the memory management service of the kernel. This task is left for future work.

5.2 Boot Sequence of OSP

Figure 8 illustrates the modified boot sequence used when launching OSP. We assume that each bootloader verifies the integrity of the succeeding bootloader using a secure boot mechanism so that we can trust the code and initial states of the OSP software components. The OSP core should start while running in the kernel mode of the secure world to access the privileged system control registers related to TrustZone and virtualization. First, it enables SMC instructions and sets the SMC call handlers for OSP. It also initializes TZASC and GIC. Next, it prepares the OSP hypervisor and the TEE by initializing virtualization features, such as the extended page tables and programming interfaces. In a multi-core environment, as each core has an independent execution environment, the OSP core wakes the other cores and initializes them as well. The OSP core finally deactivates itself by executing a secure monitor call and transfers control to the kernel. The remaining cores, apart from the primary one, jump to the idling code, a.k.a. a boot monitor.

6 Evaluation

In this section, we evaluate OSP by analyzing its performance overhead and security. Experiments were conducted on ODROID-XU3 Lite [27], which has an Exynos-5422 SoC with an ARM Cortex-A15 1.8 GHz quad-core processor and 2 GB of DRAM, used on a variant of a Samsung Galaxy S5 smartphone. The OS is Android 4.4.2 with Linux Kernel 3.10.

To obtain accurate results, we leave the device idle and let it cool down between experimental trials. Without such measures, the processor would be throttled by heat, ultimately leading to an inaccurate evaluation. Note that mobile processors can overheat within seconds if they are fully utilized. A cooling fin and fan were attached onto the target experiment board as additional countermeasures.

6.1 Performance impact

To investigate the performance impact on the system, we tested the three following cases:

- **baseline**: with a bare Android without OSP
- **hyp_on**: with OSP while the OSP hypervisor is activated
- **hyp_off**: with OSP while the OSP hypervisor is deactivated

Note that the performance of **hyp_on** represents the performance of applications when SCCs are running in other cores. We show the results normalized by the values of the **baseline**.

We experimented with popular mobile benchmarks: AnTuTu, BaseMark and Geekbench. We also experimented with other synthetic workload benchmarks with various categories: CPU and memory (Vellamo-Machine, CF-bench), JavaScript for web browsers (Vellamo-Browser), file system throughput (IOZone), graphics throughput (GFXBench) and kernel system calls (lmbench).

Figure 9 shows the experimental results. In the figure, higher values represent shorter latency times or higher throughput, where 1 represents the performance of the **baseline**. In addition, geomean indicates the normalized geometric mean values of all benchmark results.

When the OSP hypervisor is activated, performance is degraded mainly due to a high TLB miss penalty caused by complicated address translations. Therefore, memory-intensive tasks show somewhat higher overhead than computation-intensive tasks in **hyp_on**. Note that most test cases, with the exception of "DVFS Off" of AnTuTu, allow the kernel to freely adjust the CPU frequency, as in most mobile devices. Also, note that "DVFS Off" represents the case in which the frequency of the CPU is fixed at 1.2GHz such that we can eliminate the possibility of performance throttling, which could be caused by overheating or by the variances induced by DVFS itself. These results suggest that such effects do not arise.

The slowdown in **hyp_off** is near 0 % (mean = 1.003); on the other hand **hyp_on** shows visible degradation (mean = 1.066). These results reflect the effectiveness of on-demand activation for reducing hypervisor-induced overhead. Overall, OSP virtually does not incur any slowdown for the system when no SCCs are running.

6.2 World switching latency

We investigated the latency of a single round of world switch between the normal world and the OSP world. The latency depends on whether or not the OSP hypervisor is activated. Our results show that the latency is only 550 cycles when the OSP hypervisor is activated. In contrast, the world switching latency is 127,453 cycles (71 μs at 1.8 GHz), which includes 11,191 cycles to set up the OSP world in the OSP core, 550 cycles to enter and exit the OSP world, 31,450 cycles to clean and invalidate the cache memory, and 68,329 to verify the configuration of the system MMU to defeat DMA attacks. This amount of latency is likely to be tolerable in a commercial device. Note that the latency of OSP is comparable to the context switch latency of ARM processors of previous generations [22, 1].

6.3 Application benchmarks

To investigate the feasibility of OSP, we ported two applications, the Chromium web browser and a file encryptor. As a result, we confirmed that (1) OSP incurs no noticeable delay when SCCs are called infrequently, and (2) on-demand activation makes OSP effective against hypervisor overhead.

Chromium web browser Modern web browsers internally provide an autocomplete function for user convenience, which adds IDs and passwords to a login form using saved values. However, this function introduces a risk that the saved list of IDs and passwords can be exposed to untrusted software. Therefore, the list in the autocomplete function must be secured.

We conducted an experiment on the Chromium web browser for Android, version 46.0.2469.0. If the browser finds a login form, it provides the autocomplete functionality by using the `LoginDatabase` class, including `AddLogin`, `UpdateLogin` and `GetLogins` members. However, as, in the target version of the browser, the `LoginDatabase` class saves and secures IDs and passwords by simply encoding them with UTF-8, we modified it to encrypt IDs and passwords before saving them

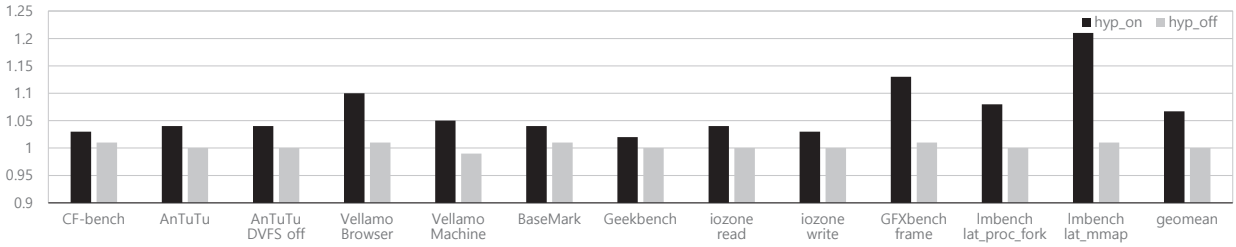


Figure 9: Performance results of OSP relative to the baseline

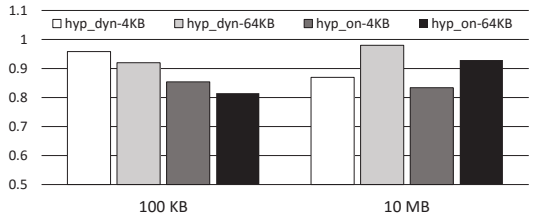


Figure 10: Performance results of the encryption SCC for 100 KB and 10 MB files relative to the baseline

(**baseline**). To evaluate OSP, instead of that, we modified the `LoginDatabase` class to save IDs and passwords after encrypting them with our SCC. Moreover, according to the lifecycle model of an SCC in Figure 3, we inserted the registration and unregistration routines of the SCC into the constructor and destructor of the `LoginDatabase`.

After making such modifications , we visited the login page of Facebook. In Chromium, the page load function and the autocomplete function run in separate threads. Therefore, we measured the page loading time and the autocompletion time separately. The experiment was repeated 100 times. The page load time averaged at 995.7 ms for both the **baseline** function and that with an SCC. Regarding the autocompletion time, the trial with an SCC averaged at 0.101 ms, which is 20.4 times slower than the **baseline**. However, when combined with the page load time, the difference in the autocompletion time is negligible.

File encryptor Many recent applications encrypt their sensitive data, such as chat logs and private pictures, for data protection purposes. In the figure, a higher value represents a longer execution time. A significant issue associated with such an approach is the method used to protect their encryption key. To address this, we implemented an SCC which provides AES-256 encryption and decryption functions and linked it to our own file encryptor using JNI. The file encryptor reads a file in chunks of 4 KB and 64 KB and encrypts each chunk with the SCC.

Figure 10 shows experimental results, consisting of the results of 100 separate executions of different input file sizes. In comparison to the **baseline**, **hyp_dyn** (enabling the on-demand feature), shows a degraded performance level due to activation overhead, proportional to the number of invocations of the SCC. We investigate the impact of on-demand activation by comparing **hyp_dyn** to **hyp_on** (disabling the on-demand feature). As a result, **hyp_dyn** is more efficient than **hyp_on** despite the accumulated overhead from on-demand activations. This is likely due to the fact that **hyp_on** incurs performance overhead caused by the hypervisor while the host application completes file operations between the SCC calls.

6.4 Security analysis

6.4.1 TCB size

In OSP, the OSP hypervisor is the TCB of SCCs and the OSP core residing in the secure world belongs to the TCB of the entire system. To estimate the safety of OSP in terms of TCB size, we measured the number of source lines of our OSP prototype with the SLOCCount tool [57]. The OSP hypervisor consists of $<$ 3,000 C SLOC and $<$ 500 assembly SLOC. The OSP core has $<$ 700 C SLOC and $<$ 100 assembly SLOC. In conclusion, OSP has as small a TCB as in previous works [47, 36, 62].

6.4.2 On-demand activation

After the system is turned on, OSP undergoes various state transitions. We analyzed the security of OSP as follows according to its states.

Initialization. The initialization of OSP is carried out as part of the boot sequence described in Section 5.2. We guarantee its safety under two assumptions: first, there must be no exploitable vulnerabilities in the code of OSP; second, a secure boot mechanism must be implemented. Therefore, we can be sure that the loaded OSP code and initial states are intact and that the boot stage of OSP is completed with certain known good states.

Hyp_off. While the OSP hypervisor is not deployed, the OSP core temporarily covers the OSP world with the secure world by configuring TZASC. All code and data residing in the OSP world are isolated from the normal world by the TrustZone components that are scattered across the system so that malicious memory accesses from both untrusted software in the normal world and misconfigured peripherals are completely prevented.
Hyp_on. While the OSP hypervisor is deployed, the OSP world strictly belongs to the normal world within the concept of TrustZone, which splits all system resources into the normal and secure worlds. Nevertheless, the OSP world is still secure from untrusted software, as the OSP hypervisor is capable of blocking unallowed memory accesses to the world by means of extended paging. DMA attacks are also thwarted by examining the mapping tables of the system mmu so as to prevent the OSP world from being exposed to peripherals.

6.4.3 Malicious SCC

As OSP allows application developers to deploy their SCCs without thorough verification or examination, it is reasonable to postulate that there could be SCCs built with malicious intention. Malicious SCCs might attempt to tamper with the normal world software or other SCCs. In OSP, however, because each SCC is strongly isolated, the attack surface is minimized, leaving few windows for malicious SCCs to compromise the hypervisor. They may try to abuse the service interface of OSP with crafted parameters to compromise the OSP hypervisor. To defeat such approaches, we may need to execute each SCC in a sandbox [59, 33]. Unfortunately, as the current OSP prototype does not contain such defense mechanisms, there is a possibility that the OSP hypervisor could be compromised. Nevertheless, as the OSP core, being located in the secure world, still has full control over the OSP hypervisor, it can ensure the integrity of the OSP hypervisor by using TrustZone-based solutions [9, 25].

7 Future work

Trusted I/O path. Although the currently implemented OSP does not offer a trusted I/O path, it is a desired feature for application developers. With this feature, SCCs could directly interact with users without going through the vulnerable OS kernel. Fortunately, as explained in 2.1, TrustZone includes various components that are capable of isolating interrupts and bus transactions between the CPU cores and peripherals. They are sufficient to facilitate the implementation of a trusted I/O path [32, 51]. Similar to academic works, off-the-shelf mobile devices are known to depend on TrustZone to implement a trusted path on fingerprint sensors or other components.

Therefore, it would be reasonable for OSP to reley on the TrustZone-based trusted I/O path to provide features for SCCs rather than to develop its own trusted I/O path. Therefore, OSP initially needs to establish a secure channel between the OSP world and the secure world. The OSP core can do this by creating and passing a session key to the OSP hypervisor early in the boot stage. The OSP hypervisor can then offer a trusted I/O path, which is implemented in TrustZone, to SCCs through the OSP core without the intervention of the OS kernel.
Kernel-mode SCC. In this paper, we have assumed that SCCs are executed in the user mode. However, we can also consider other SCCs running in the kernel mode. For example, if there are SCCs that are intended to monitor the integrity of the kernel, they must run in the kernel mode to execute privileged instructions or to access privileged data structures such as page tables. We believe that kernel-mode SCCs are difficult to protect in TrustZone-based solutions due to the high security risk involved in permitting privilege instructions. On the other hand, we deem that OSP could cover such SCCs as well. To provide isolated execution environments, OSP depends on the OSP hypervisor working beneath the OS kernel; therefore, we can improve the OSP hypervisor to mediate and verify behaviors of kernel-mode SCCs that may corrupt the system.
Other Future Work. ARM introduced the big.LITTLE architecture, which leverages big (high-performance) cores or little (low-performance) cores depending on the performance requirements of tasks, thereby improving the power efficiency. The current prototype of OSP has yet to support this technique. However, as it becomes more popular on mobile devices, it will be necessary to upgrade OSP to support it.

8 Related work

In this section, we compare OSP with existing solutions attempting to protect software.
TrustZone-based solutions TrustZone, originated by ARM, has been spotlighted as a secure and lightweight solution to protect SCCs. Recently, it was also adopted in the x86 architecture by AMD. To enhance its usability, TLR [45, 44] ported the .NET framework inside TrustZone so that it enables SCCs programmed with the .NET bytecode to execute in the secure world. However, its monolithic design in which all SCCs are sharing a single world will increase the attack surface in proportion to the number of installed SCCs. TrustICE [52] addressed this problem by providing each SCC with a separated execution environment, called ICE, in the normal world. Thus, in that work, third-party developers are permitted to pro-

tect their SCCs. This work achieves a similar objective as OSP, but unlike OSP, TrustICE was designed to create isolated execution environments based on the kernel mode, thus using the same privilege level with the untrusted OS kernel. Consequently, it faces limitations in supporting multi-core environments. While an ICE runs in a core, the other cores must be suspended until the ICE is terminated, to prevent the OS kernel from accessing the ICE.

Hypervisor-based solutions Many studies have attempted to provide isolated execution environments by leveraging hypervisors. TERRA [24] and Proxos [55] attempted to provide each application with its own operating system. Overshadow [17] and SP^3 [58] protected application data from being exposed to untrusted OSes by encrypting the data transparently.

As frequent encryption operations incur significant performance overhead in the system, numerous studies have constructed isolated execution environments using elaborate access-control mechanisms based on the extended paging technique. InkTag [28], AppShield [18] and AppSec [43] concentrated on shielding all applications from an untrusted OS. On the other hand, TrustVisor [36], MiniBox [33] and Wimpy Kernel [62] focused on protecting security critical portions of applications (SCCs in this paper) rather than all applications. These solutions attempt to reduce the code size of their hypervisors in order to reduce the size of their TCB as well. Our OSP can be considered similar to these solutions in the sense that OSP protects a small portion of an application using a lightweight hypervisor, minimizing the code size of its hypervisor.

However, all of the aforementioned techniques place a burden on the system through persistent computational overhead for their hypervisors to maintain virtualization. Curtailing such overhead by dynamically activating hypervisors was originally proposed in earlier studies [35, 41], in which the technique was used for efficient OS maintenance based on a hypervisor. However, these techniques are inadequate for software protection as their designs do not consider security constraints. Meanwhile, P-MAPS [42] and another study [60] adopted aforementioned techniques for security purposes. However, in comparison with OSP, their implementations are not lightweight because they rely on time-consuming cryptographic functions of TPM for on-demand functionality. In particular, P-MAPS has world-switching latency of 300 ms, which is clearly noticeable to users.

Hardware-based Approaches AEGIS [50], Bastion [15], SecureME [19] and XOMOS [34] provide secure execution environments. However, they are not compatible with conventional systems because they require new architectural features.

Flicker [37], based on TPM and DRTM of Intel x86, supports on-demand protections for SCCs. Similar to P-MAPS, Flicker incurs a world switching latency problem owing to its dependency of TPM. SICE [10] and Secure Switch [53] create isolated execution environments with an additional CPU mode known as the system management mode (SMM). However, SICE shows a few seconds of latency when entering the isolated execution environment, and Secure Switch can only build secure environments in a specific type of memory, i.e., SMRAM, which is physically limited.

Intel recently proposed Software Guard Extension (SGX) [30], containing a new set of special instructions for creating isolated execution environments. SGX is secure against various memory attacks [20] including cold-boot attacks [26] and bus monitoring attacks [49, 54]. Although it is not yet deployed in commercial products, HAVEN [11] and VC3 [46] demonstrated its effectiveness through real-world scenarios in a cloud system. Unfortunately, available of this technique is limitedly to the Intel x86 architecture.

9 Conclusion

In this paper, we have proposed OSP, a TrustZone-hypervisor hybrid protection system, which aims to provide isolated computing environments for SCCs in an efficient and secure manner. OSP reduces the virtualization overhead by leveraging the on-demand hypervisor activation scheme that is efficiently carried out with assistance of TrustZone. To measure the performance of OSP on a mobile device, we performed a set of experiments with ODROID-XU3-Lite using the mobile processor adopted by latest commercial smartphones. Our evaluations have shown that OSP achieves very low performance overhead during hypervisor hibernation (near 0 %) and efficiently protects SCCs with low activation latency ($< 100\ \mu$s).

Acknowledgments. We thank our shepherd Theodore Ts'o, the reviewers and Kang G. Shin for their insightful remarks that improved the paper. This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0190-16-2010, Development on the SW/HW modules of Processor Monitor for System Intrusion Detection), the National Research Foundation of Korea(NRF) grant funded by the Korea government (MSIP) (No. 2014R1A2A1A10051792), the MSIP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-R0992-16-1006) supervised by the IITP(Institute for Information & communications Technology Promotion), Inter-University Semiconductor Research Center (ISRC) and the Brain Korea 21 Plus Project in 2016.

References

[1] Performance Measurement on ARM. http://www.pengutronix.de/development/kernel/arm-benchmarks-20100729_en.html.

[2] ALVES, T., AND FELTON, D. Trustzone: Integrated hardware and software security. *ARM white paper* (2004).

[3] ARM. System memory management unit (smmu). http://www.arm.com/products/system-ip/controllers/system-mmu.php.

[4] ARM. Primecell infrastructure amba3 trustzone memory adapter (bp141). In *ARM DTO 0017A* (2004).

[5] ARM. Primecell infrastructure amba3 trustzone protection controller (bp147). In *ARM DTO 0015A* (2004).

[6] ARM. Architecture reference manual armv7-a and armv7-r edition. In *DDI 0406C* (2009).

[7] ARM. Trustzone address space controller (tzc-380). In *ARM DDI 0431B* (2010).

[8] ARM. Generic interrupt controller architecture version 2.0. In *ARM IHI 0048B* (2013).

[9] AZAB, A. M., NING, P., SHAH, J., CHEN, Q., BHUTKAR, R., GANESH, G., MA, J., AND SHEN, W. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014).

[10] AZAB, A. M., NING, P., AND ZHANG, X. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)* (2011).

[11] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of 11th USENIX Symposium on Operating Systems Design and Implementation* (2014).

[12] BHARGAVA, R., SEREBRIN, B., SPADINI, F., AND MANNE, S. Accelerating two-dimensional page walks for virtualized systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[13] BICKFORD, J., O'HARE, R., BALIGA, A., GANAPATHY, V., AND IFTODE, L. Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the eleventh workshop on mobile computing systems & applications* (2010).

[14] BRUMLEY, D., AND SONG, D. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium* (2004).

[15] CHAMPAGNE, D., AND LEE, R. B. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on* (2010).

[16] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[17] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[18] CHENG, Y., DING, X., AND DENG, R. Appshield: Protecting applications against untrusted operating system. *Singaport Management University Technical Report, SMU-SIS-13* (2013).

[19] CHHABRA, S., ROGERS, B., SOLIHIN, Y., AND PRVULOVIC, M. Secureme: a hardware-software approach to full system security. In *Proceedings of the international conference on Supercomputing* (2011).

[20] COLP, P. J., ZHANG, J., GLEESON, J., SUNEJA, S., DE LARA, E., RAJ, H., SAROIU, S., AND WOLMAN, A. Protecting data on smartphones and tablets from memory attacks. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2015).

[21] DALL, C., AND NIEH, J. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems (ASPLOS)* (2014).

[22] DAVID, F. M., CARLYLE, J. C., AND CAMPBELL, R. H. Context switch overheads for linux on arm platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science* (2007).

[23] DEVRIENT, G. . Mobicore. http://www.gi-de.com/en/trends_and_insights/mobicore/trusted-mobile-services.jsp.

[24] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003).

[25] GE, X., VIJAYAKUMAR, H., AND JAEGER, T. Sprobes: Enforcing kernel code integrity on the trustzone architecture.

[26] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th Conference on Security Symposium* (2008).

[27] HARDKERNEL. Odroid. http://www.hardkernel.com.

[28] HOFMANN, O. S., KIM, S., DUNN, A. M., LEE, M. Z., AND WITCHEL, E. Inktag: Secure applications on an untrusted operating system. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[29] INC, A. Secure virtual machine architecture reference manual, 2005.

[30] INTEL. Software guard extensions programming reference.

[31] LABS, M. Threats report. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf, 2015.

[32] LI, W., MA, M., HAN, J., XIA, Y., ZANG, B., CHU, C.-K., AND LI, T. Building trusted path on untrusted device drivers for mobile devices. In *Proceedings of 5th Asia-Pacific Workshop on Systems* (2014).

[33] LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. Minibox: A two-way sandbox for x86 native code. In *2014 USENIX Annual Technical Conference (ATC)* (2014).

[34] LIE, D., THEKKATH, C. A., AND HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (2003).

[35] LOWELL, D. E., SAITO, Y., AND SAMBERG, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems* (2004).

[36] MCCUNE, J. M., LI, Y., QU, N., ZHOU, Z., DATTA, A., GLIGOR, V., AND PERRIG, A. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy* (2010).

[37] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems* (2008).

[38] MISRA, S. C., AND BHAVSAR, V. C. Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Computational Science and Its ApplicationsICCSA 2003*. 2003.

[39] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal* (2006).

[40] NORDHOLZ, J., VETTER, J., PETER, M., JUNKER-PETSCHICK, M., AND DANISEVSKIS, J. Xnpro: Low-impact hypervisor-based execution prevention on arm. In *Proceedings of the 5th International Workshop on Trustworthy Embedded Devices* (2015).

[41] OMOTE, Y., SHINAGAWA, T., AND KATO, K. Improving agility and elasticity in bare-metal clouds. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015).

[42] RAVI, S., ULHAS, W., AND PRASHANT, D. Dynamic software application protection. *Intel Technology Journal* (2009).

[43] REN, J., QI, Y., DAI, Y., WANG, X., AND SHI, Y. Appsec: A safe execution environment for security sensitive applications. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2015).

[44] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Trusted language runtime (tlr): enabling trusted applications on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications* (2011).

[45] SANTOS, N., RAJ, H., SAROIU, S., AND WOLMAN, A. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[46] SCHUSTER, F., COSTA, M., FOURNET, C., GKANTSIDIS, C., PEINADO, M., MAINAR-RUIZ, G., AND RUSSINOVICH, M. Vc3: Trustworthy data analytics in the cloud using sgx. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015).

[47] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007).

[48] SIERRAWARE. Open virtualization - arm trustzone and arm hypervisor open source software. `http://www.sierraware.com`.

[49] SOLUTIONS, E. Analysis tools for ddr1, ddr2, ddr3, embedded ddr and fully buffered dimm modules. `http://www.epnsolutions.net/ddr.html`, 2014.

[50] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. Aegis: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing* (2003).

[51] SUN, H., SUN, K., WANG, Y., AND JING, J. Trustotp: Transforming smartphones into secure one-time password tokens. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015).

[52] SUN, H., SUN, K., WANG, Y., JING, J., AND WANG, H. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on* (2015).

[53] SUN, K., WANG, J., ZHANG, F., AND STAVROU, A. Secureswitch: Bios-assisted isolation and switch between trusted and untrusted commodity oses. In *Network and Distributed System Security Symposium (NDSS)* (2012).

[54] SYSTEM, F. Ddr2 800 bus analysis probe. `http://www.futureplus.com/download/datasheet/fs2334_ds.pdf`, 2006.

[55] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)* (2006).

[56] UNIFIED, E. Inc. unified extensible firmware interface specification, 2014.

[57] WHEELER, D. A. Sloccount. `http://www.dwheeler.com/sloccount`, 20015.

[58] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)* (2008).

[59] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009).

[60] YEFREMOV, D., AND IAKOVENKO, P. An approach to on the fly activation and deactivation of virtualization-based security systems. In *Proceedings of the Spring/Summer Young Researchers Colloquium on Software Engineering* (2010).

[61] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012).

[62] ZHOU, Z., YU, M., AND GLIGOR, V. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *Proceedings of the IEEE Symposium on Security and Privacy* (2014).

gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space

Mochi Xue[1,2], Kun Tian[2], Yaozu Dong[1,2], Jiacheng Ma[1], Jiajun Wang[1],
Zhengwei Qi[1], Bingsheng He[3], Haibing Guan[1]
{*xuemochi, mjc0608, jiajunwang, qizhenwei, hbguan*}*@sjtu.edu.cn*
{*kevin.tian, eddie.dong*}*@intel.com he.bingsheng@gmail.com*
[1]*Shanghai Jiao Tong University,* [2]*Intel Corporation,* [3]*National University of Singapore*

Abstract

With increasing GPU-intensive workloads deployed on cloud, the cloud service providers are seeking for practical and efficient GPU virtualization solutions. However, the cutting-edge GPU virtualization techniques such as gVirt still suffer from the restriction of scalability, which constrains the number of guest virtual GPU instances.

This paper introduces *gScale*, a scalable GPU virtualization solution. By taking advantage of the GPU programming model, gScale presents a dynamic sharing mechanism which combines partition and sharing together to break the hardware limitation of global graphics memory space. Particularly, we propose three approaches for gScale: (1) the private shadow graphics translation table, which enables global graphics memory space sharing among virtual GPU instances, (2) ladder mapping and fence memory space pool, which allows the CPU to access host physical memory space (serving the graphics memory) bypassing global graphics memory space, (3) slot sharing, which improves the performance of vGPU under a high density of instances.

The evaluation shows that gScale scales up to 15 guest virtual GPU instances in Linux or 12 guest virtual GPU instances in Windows, which is 5x and 4x scalability, respectively, compared to gVirt. At the same time, gScale incurs a slight runtime overhead on the performance of gVirt when hosting multiple virtual GPU instances.

1 Introduction

The Graphic Processing Unit (GPU) is playing an indispensable role in cloud computing as GPU efficiently accelerates the computation of certain workloads such as 2D and 3D rendering. With increasing GPU intensive workloads deployed on cloud, cloud service providers introduce a new computing paradigm called *GPU Cloud* to meet the high demands of GPU resources (e.g., Amazon EC2 GPU instance [2], Aliyun GPU server [1]).

As one of the key enabling technologies of GPU cloud, GPU virtualization is intended to provide flexible and scalable GPU resources for multiple instances with high performance. To achieve such a challenging goal, several GPU virtualization solutions were introduced, i.e., GPUvm [28] and gVirt [30]. gVirt, also known as GVT-g, is a full virtualization solution with mediated pass-through support for Intel Graphics processors. Benefited by gVirt's open-source distribution, we are able to investigate its design and implementation throughly. In each virtual machine (VM), running with native graphics driver, a virtual GPU (vGPU) instance is maintained to provide performance-critical resources directly assigned, since there is no hypervisor intervention in performance critical paths. Thus, it optimizes resources among the performance, feature, and sharing capabilities [5].

For a virtualization solution, scalability is an indispensable feature which ensures high resource utilization by hosting dense VM instances on cloud servers. Although gVirt successfully puts GPU virtualization into practice, it suffers from scaling up the number of vGPU instances. The current release of gVirt only supports 3 guest vGPU instances on one physical Intel GPU[1], which limits the number of guest VM instances down to 3. In contrast, CPU virtualization techniques (e.g., Xen 4.6 guest VM supports up to 256 vCPUs [11]) are maturely achieved to exploit their potential. The mismatch between the scalability of GPU and other resources like CPU will certainly diminish the number of VM instances. Additionally, high scalability improves the consolidation of resources. Recent studies (eg., VGRIS [26]) have observed that GPU workloads can fluctuate significantly on GPU utilization. Such low scalability of gVirt could result in severe GPU resource underutilization. If more guest VMs can be consolidated to a single host, cloud providers have more chances to multiplex the GPU power among VMs with different workload pat-

[1] In this paper, Intel GPU refers to the Intel HD Graphics embedded in HASWELL CPU.

terns (e.g., scheduling VMs with GPU intensive or idle patterns) so that the physical resource usage of GPU can be improved.

This paper explores the design of gVirt, and presents gScale, a practical, efficient and scalable GPU virtualization solution. To increase the number of vGPU instances, gScale targets at the bottleneck design of gVirt and introduces a dynamic sharing scheme for global graphics memory space. gScale provides each vGPU instance with a private shadow *graphics translation table* (GTT) to break the limitation of global graphics memory space. gScale copies vGPU's private shadow GTT to physical GTT along with the context switch. The private shadow GTT allows vGPUs to share an overlapped range of global graphics memory space, which is an essential design of gScale. However, it is non-trivial to make the global graphics memory space sharable, because global graphics memory space is both accessible to CPU and GPU. gScale implements a novel *ladder mapping* mechanism and a fence memory space pool to let CPU access host physical memory space serving the graphics memory, which bypasses the global graphics memory space. At the same time, gScale proposes *slot sharing* to improve the performance of vGPUs under a high density of instances.

This paper implements gScale based on gVirt, which is comprised of about 1000 LoCs. The source code is now available on Github[2]. In summary, this paper overcomes various challenges, and makes the following contributions:

- A private shadow GTT for each vGPU, which makes the global graphics memory space sharable. It keeps a specific copy of the physical GTT for the vGPU. When the vGPU becomes the render owner, its private shadow graphics translation table will be written on the physical graphics translation table by gScale to provide correct translations.

- The ladder mapping mechanism, which directly maps guest physical address to host physical address serving the guest graphic memory. With ladder mapping mechanism, the CPU can access the host physical memory space serving the guest graphic memory, without referring to the global graphics memory space.

- Fence memory space pool, a dedicated memory space reserved in global graphics memory space with dynamic management. It guarantees that the fence registers operate correctly when a certain range of global graphics memory space is unavailable for CPU.

- Slot sharing, a mechanism to optimize the performance of vGPUs, reduces the overhead of private shadow GTT copying under a high instance density.

- The evaluation shows that gScale can provide 15 guest vGPU instances for Linux VMs or 12 guest vGPU instances for Windows VMs on one physical machine, which is 5x and 4x scalability respectively, compared to gVirt. It achieves up to 96% performance of gVirt under a high density of instances.

The rest of paper is organized as follows. Section 2 describes the background of gScale, and Section 3 reveals gVirt's scalability issue and its bottleneck design. The detailed design and implementation of gScale are presented in Section 4. We evaluate gScale's performance in Section 5 with the overhead analysis. We discuss the applicability of our work in Section 6 and the related work is in Section 7. Finally, in Section 8 we conclude our work with a brief discussion of future work.

2 Background and Preliminary

GPU Programming Model Driven by high level programming APIs like OpenGL and DirectX, graphics driver produces GPU commands into primary buffer and batch buffer while GPU consumes the commands accordingly. The primary buffer is designed to deliver the primary commands with a ring structure, but the size of primary buffer is limited. To make up for the space shortage, batch buffer is linked to the primary buffer to deliver most of the GPU commands.

GPU commands are produced by CPU and transferred from CPU to GPU in batches. To ensure that GPU consumes the commands after CPU produces them, a notification mechanism is implemented in the primary buffer with two registers. The *tail* register is updated when CPU finishes the placement of commands, and it informs GPU to get commands in the primary buffer. When GPU completes processing all the commands, it writes the *head* register to notify CPU for incoming commands [30].

Graphics Translation Table and Global Graphics Memory Space Graphics translation table (GTT), sometimes known as *global graphics translation table*, is a memory-resident page table providing the translations from logical graphics memory address to physical memory address, as Figure 1 shows. It is worth noting that the physical memory space served by GTT is also assigned to be the global graphics memory space, especially for GPUs without dedicated memory, such as Intel's GPU. However, through the Aperture [6], a range defined in the graphics memory mapping input/output (MMIO), CPU could also access the global graphics memory space.

[2]https://github.com/inkpool/XenGT-Preview-kernel/tree/gScale

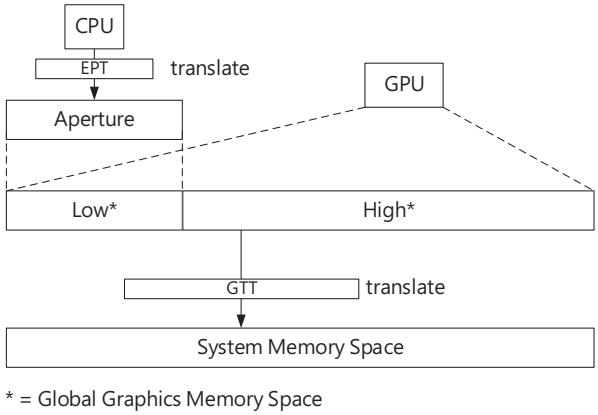


Figure 1: Graphics Translation Table

And this CPU's visible part of global graphics memory is called *low global graphics memory*, while the rest part is called *high global graphics memory*. To be specific, Intel GPU has a 2MB GTT which maps to a 2GB graphics memory space. The aperture range could maximally be 512KB which maps to 512MB graphics memory space visible by CPU. Accordingly, the low graphics memory space is 512MB, while the high graphics memory space is 1536MB.

gVirt gVirt is the first product-level full GPU virtualization solution with mediated pass-through [30]. So the VM running native graphics driver is presented with a full featured virtualized GPU. gVirt emulates virtual GPU (vGPU) for each VM, and conducts context switch among vGPUs. vGPUs are scheduled to submit their commands to the physical GPU continuously, and each vGPU has a 16ms time slice. When time slice runs out, gVirt switches the render engine to next scheduled vGPU. To ensure the correct and safe switch between vGPUs, gVirt saves and restores vGPU states, including internal pipeline state and I/O register states.

By passing-through the accesses to the frame buffer and command buffer, gVirt reduces the overhead of performance-critical operations from a vGPU. For global graphics memory space, resource partition is applied by gVirt. For local graphics memory space, gVirt implements per-VM local graphics memory [30]. It allows each VM to use the full local graphics memory space which is 2GB in total. The local graphics memory space is only accessible to vGPU, so gVirt can switch the graphics memory spaces among vGPUs when switching the render ownership.

3 Scalability Issue

The global graphics memory space can be accessed simultaneously by CPU and GPU. gVirt has to present VMs with their global graphics memory spaces at any time, leading to the *resource partition*. As shown in Figure 2, when a vGPU instance is created with a VM, gVirt only assigns a part of host's low global graphics memory and a part of host's high global graphics memory to the vGPU, as its *low* global graphics memory and *high* global graphics memory, respectively. These two parts together comprise the vGPU's global graphics memory space. Moreover, the partitioned graphics memory spaces are mapped by a shared shadow GTT, which is maintained by gVirt to translate guest graphics memory address to host physical memory address.

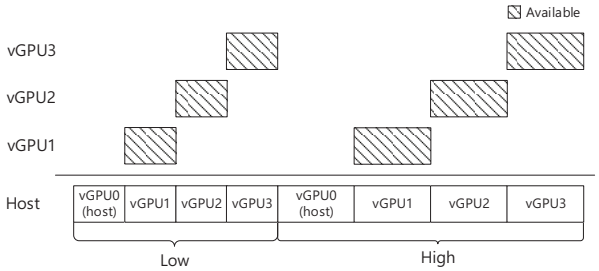


Figure 2: Global Graphics Memory Space with Partition

To support simultaneous accesses from VMs, the shared shadow GTT has to carry translations for all the VMs, which means the guest view of shadow GTT is exactly the same with host's, as shown in Figure 5. gVirt introduces an address space ballooning mechanism to balloon the space that does not belong to the VM. gVirt exposes the partition information to VM's graphics driver, and graphics driver marks the space which does not belong to the VM as "ballooned" [30]. Note here, gVirt's memory space ballooning mechanism is for resource isolation, which is different from traditional memory ballooning technique [31]. Though guests have the same view of shadow GTT with the host, with ballooning mechanism, guest VM can only access the global graphics memory space asigned to itself.

Due to the *resource partition* mechanism for global graphics memory space, with a fixed size of global graphics memory, the number of vGPUs hosted by gVirt is limited. If gVirt wants to host more vGPUs, it has to configure vGPUs with less global graphics memory. However, it sacrifices vGPU's functionality if we increase the number of vGPUs by shrinking the global graphics memory size of vGPUs. Moreover, the graphics driver reports errors or even crashes when it cannot allocate memory from global graphics memory space [4]. For instance, a vGPU with deficient global graphics

memory size may lose functionality under certain workloads which need the high requirements of global graphics memory. In fact, more global graphics memory does not bring performance improvement for vGPUs, because global graphics memory only serves frame buffer and ring buffer with limited sizes, while the massive rendering data resides in local graphics data [30]. Specifically, for vGPU in Linux VM, the 64MB low global graphics memory and 384MB high global graphics memory are recommended. For vGPU in Windows VM, the recommend configuration is 128MB low global graphics memory and 384MB high global graphics memory [12]. In the scalability experiment of gVirt [30], it hosted 7 guest vGPUs in Linux VMs. However, the global graphics memory size of vGPU in that experiment is smaller than the recommended configuration. Such configuration cannot guarantee the full functionality of vGPU, and it would incur errors or crashes for vGPU under certain workloads because of the deficiency of graphics memory space [4]. In this paper, the vGPUs are configured with recommended configuration.

Actually, the current source code (2015Q3) of gVirt sets the maximal vGPU number as 4. For platform with Intel GPU, there is 512MB low global graphics memory space and 1536MB high global graphics memory space in total. While gVirt can only provide 3 guest vGPUs (64MB low global graphics memory, and 384MB high global graphics memory) for Linux VMs or 3 guest vGPUs (128MB low global graphics memory, and 384MB high global graphics memory) for Windows VMs, because the host VM also occupies one vGPU. As a GPU virtualization solution, gVirt is jeopardized by its scalability issue. The static partition of global graphics memory space is the root cause of the scalability issue. In this paper, we attempt to break the limitation of static resource partition and sufficiently improve the scalability for gVirt.

4 Design and Implementation

The architecture of gScale is shown in Figure 3. To break the limitation of global graphics memory, gScale proposes a dynamic sharing scheme which combines partition and sharing together as Figure 4 illustrates. For the access of GPU, we introduce private shadow GTT to make global graphics memory space sharable. For the access of CPU, we present ladder mapping to allow CPU to directly access host physical memory space serving the graphics memory, which bypasses the global graphics memory space. For concurrent accesses of CPU and GPU, gScale reserves a part of low global graphics memory as *the fence memory space pool* to ensure the functionality of fence registers. gScale also divides the high global graphics memory space into several slots to leverage the overhead caused by private shadow GTT copying.

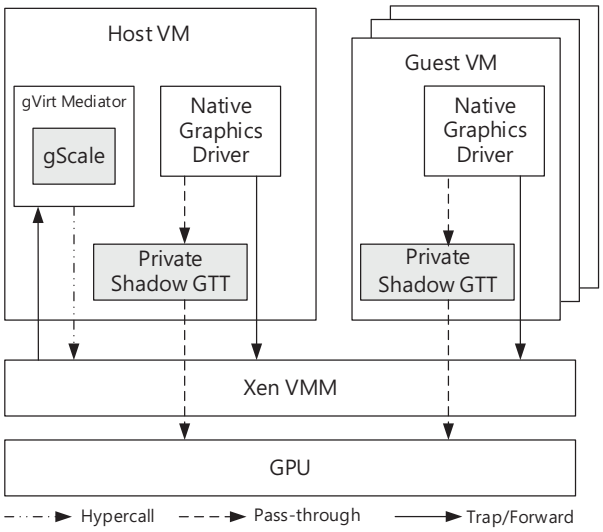


Figure 3: Architecture

In this section, the design of gScale addresses three technical challenges: (1) how to make global graphics memory space sharable among vGPUs, (2) how to let CPU directly access host memory space serving the graphics memory, which bypasses global graphics memory space, and (3) how to improve the performance of vGPUs under a high instance density.

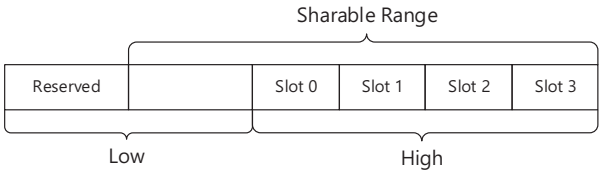


Figure 4: Dynamic Sharing Scheme of gScale

4.1 Private Shadow GTT

It is a non-trivial task to make the global graphics memory space sharable among vGPUs, for that CPU and GPU access the low global graphics memory space simultaneously, as we mentioned in Section 2. However, high global graphics memory space is only accessible to GPU, which makes it possible for vGPUs to share high global graphic memory space. Taking advantages of GPU programming model, vGPUs are scheduled to take turns to be served by render engine, and gVirt conducts context switch before it changes the ownership of render engine. This inspires us to propose the private shadow GTT for each vGPU.

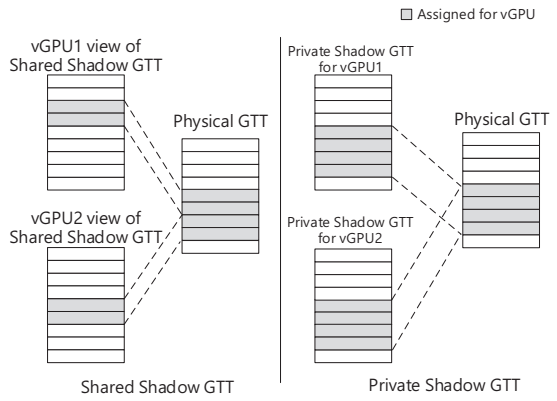


Figure 5: Private Shadow GTT

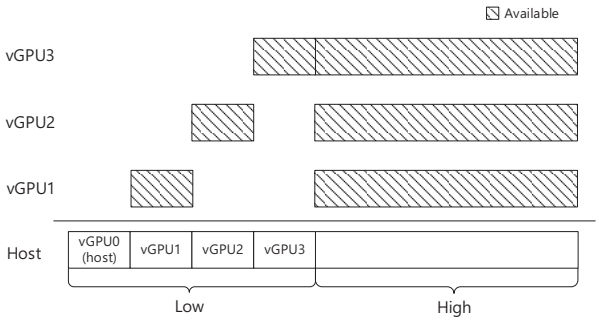


Figure 6: Sharable Global Graphics Memory Space

Figure 5 shows the gVirt's shared shadow GTT and gScale's private shadow GTT. Specifically, shared shadow GTT is introduced to apply the resource partition on global graphics memory space. It provides every vGPU with a same view of physical GTT, while each vGPU is assigned with a different part of shadow GTT. Accordingly, each vGPU occupies the different ranges of global graphics memory space from others. However, gScale's private shadow GTT is specific for each vGPU, and it provides vGPU with a unique view of global graphics memory space. Moreover, the translations that private shadow GTT contains are only valid for its corresponding vGPU. And gScale copies vGPU's private shadow GTT onto the physical GTT along with the context switch to ensure that translations of physical GTT are correct for the upcoming vGPU. When vGPU owns the physical engine, gScale synchronizes the modifications of physical GTT to vGPU's private shadow GTT.

By manipulating the private shadow GTTs, gScale could allow vGPUs to use an overlapped range of global graphics memory, which makes the high global graphics memory space sharable, as shown in Figure 6. However, low graphics memory space is still partitioned among the vGPUs, for that it is also visible to CPU. Simply using private shadow GTT to make low graphics memory space sharable would provide vCPU with wrong translations.

On-demand Copying Writing private shadow GTT onto physical GTT incurs the overhead. gScale introduces *on-demand copying* to reduce unnecessary copying overhead. Currently, gScale is able to assign the whole sharable global graphics memory space to a vGPU. Instead, gScale only configures vGPU with the sufficient global graphics memory, for that more global graphics memory does not increase the performance of vGPU while it could increase the overhead of copying shadow GTT. Although the size of private GTT is exactly the same with physical GTT, vGPU is configured with a portion of available global graphics memory space (corresponding to only part of vGPU's private shadow GTT). By taking advantage of this characteristic, gScale only copies the demanding part of vGPU's private shadow GTT to the physical GTT, which mitigates the unnecessary overhead.

4.2 Ladder Mapping

It is not enough to only make high global graphics memory space sharable because the static partition applied to low global graphics memory space still constrains the number of vGPUs. Low global graphics memory space is accessible to both CPU and GPU, while CPU and GPU are scheduled independently. gScale has to present VMs with their low global graphics memory spaces at all time. Intel GPU does not have dedicated graphics memory, while the graphics memory is actually allocated from system memory. The graphics memory of VM actually resides in host physical memory. gScale proposes *the ladder mapping* to allow CPU to directly access the host memory space serving the graphics memory which bypasses the global graphics memory space.

When a VM is created, gScale maps VM's guest physical memory space to host physical memory space by Extended Page Table (EPT). EPT is a hardware supported page table for virtualization, which translates guest physical address to host physical address [23]. Through the aperture, a range of MMIO space in host physical memory space, CPU could access the low part of global graphics memory space. With the translations in GTT, the global graphics memory address is translated into host physical address serving the graphics memory. Finally, CPU could access the graphics data residing in host physical memory space.

Figure 7 shows the initial mapping we mentioned above, and through the Step 1, 2 and 3, guest physical address is translated into host physical address. When the process is completed, a translation between guest physical address and host physical address serving the graph-

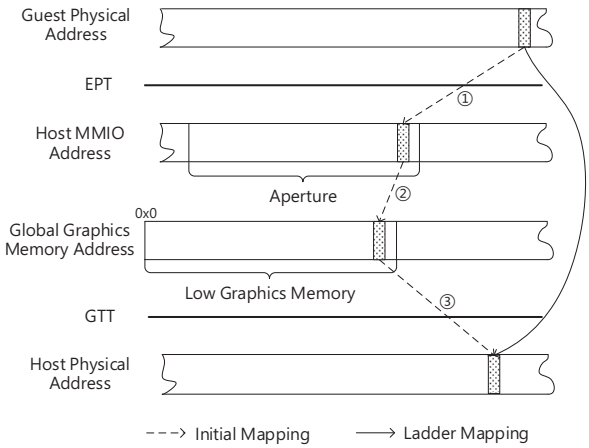


Figure 7: Ladder Mapping

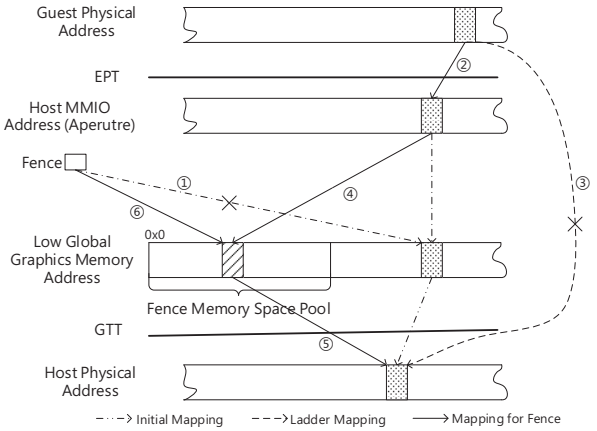


Figure 8: Fence Memory Space Pool

ics memory is established. After that, gScale modifies the translation of EPT to directly translate the guest physical address to host physical address serving the graphics memory without the reference of global graphics memory address. We call this mechanism the ladder mapping, which is constructed when CPU accesses global graphics memory space by referring to the GTT. gScale monitors the GTT at all time, and builds ladder mapping as long as the translation of GTT is modified by CPU. In a nutshell, the ladder mapping is to allow CPU to access host memory space bypassing the global graphics memory space. After that, gScale could make low global graphics memory space sharable with private shadow GTT.

Fence Memory Space Pool Although we use ladder mapping to force CPU to bypass the global graphics memory space, there is one exception that CPU could still access global graphics memory space through *fence registers*. Fence register contains the information about tiled formats for a specific region of graphics memory [6]. When CPU accesses this region of global graphics memory recorded in a fence register, it needs the format information in the fence to operate the graphics memory. However, after we enable ladder mapping, the global graphics memory space is no longer available for CPU. The global graphics memory address in fence register is invalid for CPU.

To address the malfunction of fence registers, gScale reserves a dedicated part of low global graphics memory to work for fence registers, and enables dynamic management for it. We call this reserved part of low global graphics memory, the *fence memory space pool*. Figure 8 illustrates the workflow of how fence memory space pool works:

Step 1, when a fence register is written by graphics driver, gScale acquires the raw data inside of the register. By analyzing the raw data, gScale gets the format information and the global graphics memory space range served by this fence register.

Step 2, by referring to the initial mapping of EPT, gScale finds the guest physical memory space range which corresponds to the global graphics memory space range in the register. Though the initial mapping of EPT is replaced by ladder mapping, it is easy to restore the original mapping with a backup, because the initial mapping is continuous with clear offset and range [6]. After that, this range of guest physical memory space is again mapped to a range of physical memory space within the aperture.

Step 3, gScale suspends the ladder mapping for this range of guest physical memory space, and allocates a range of memory space in the fence memory space pool with same size.

Step 4, gScale maps the host physical memory space in aperture to the memory space newly allocated in fence memory space pool.

Step 5, gScale copies the entries of GTT serving the graphics memory space in fence register to the part of GTT corresponding to the graphics memory space newly allocated in fence memory space pool.

Step 6, gScale writes the new graphics memory space range along with untouched format information into the fence register. To this end, gScale constructs a temporary mapping for fence register, and CPU could finally use the information in fence register correctly.

When a fence register is updated, gScale restores the ladder mapping for the previous range of global graphics memory space that fence register serves, and frees its corresponding memory space in the fence memory space pool. After that, gScale repeats the procedure as we mentioned above to ensure the updated register work correctly with fence memory space pool.

4.3 Slot Sharing

In real cloud environments, the instances hosted by cloud may not remain busy at all time, while some instances become idle after completing their tasks [24]. gScale implements *slot sharing* to improve the performance of vGPU instance under a high instance density. Figure 9 shows the layout of physical global graphics memory space, gScale divides the high global graphics memory space into several slots, and each slot could hold one vGPU's high graphics memory. gScale could deploy several vGPUs in the same slot. As we mentioned in Section 2, high global graphics memory space provided by Intel GPU is 1536MB, while 384MB is sufficient for one VM. However, gScale only provides slots for VMs in high graphics memory space, for that the amount of low global graphics memory space is 512MB which is much smaller than high global graphics memory space. There is no free space in low graphics memory space spared for slots.

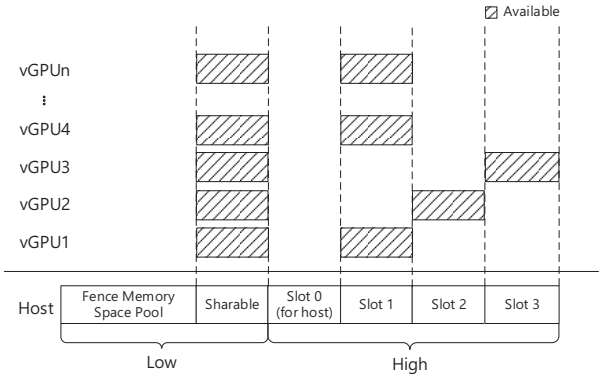


Figure 9: Slot Sharing

Optimized Scheduler gScale does not conduct context switch for idle vGPU instances, which saves the cost of context switch and private shadow GTT copying. For vGPU instances without workloads, they do not submit commands to physical engine. gScale skips them, and focuses on serving the instances with heavy workloads. At the same time, gScale does not copying entries from idle vGPU's private shadow GTT to physical GTT. With slot sharing, if there is only one active vGPU in a slot, this vGPU will own the slot. gScale keeps its high global memory part of private shadow GTT on physical GTT without entry copying. With optimized scheduler, slot sharing could effectively reduce the overhead of private shadow GTT copying, and we have a micro overhead analysis in Section 5.4.

gScale currently has 4 slots (1536MB/384MB = 4): one is reserved for host vGPU, while the rest 3 are shared by guest vGPUs. Slot sharing helps gScale improve guest vGPU's performance under a high instance density while only a few vGPUs are busy. We believe slot sharing could be utilized if the cloud provider deploys the guest VMs meticulously. For example, cloud providers let a busy vGPU share one slot with a few idle vGPUs.

5 Evaluation

In this section, we evaluate the scalability of gScale when it hosts an increasing number of guest vGPUs with GPU workloads. gScale scales well for GPU intensive workloads, which achieves up to 81% performance of gVirt when it scales to 15 vGPUs. We compare the performance of gScale with gVirt, and it turns out gScale brings negligible performance trend. Also, the performance of gScale and its basic version (without slot sharing) under a high density of instances is compared. In our experiments, slot sharing improves the performance of gScale up to 20%, and mitigates the overhead caused by copying private shadow GTT entries up to 83.4% under certain circumstances.

5.1 Experimental Setup

Host Machine Configuration	
CPU	Intel E3-1285 v3 (4 Cores, 3.6 GHz)
GPU	Intel HD Graphics P4700
Memory	32GB
Storage	SAMSUNG 850Pro 256GB * 3
Host VM Configuration	
vCPU	4
Memory	3072 MB
Low Global GM	64 MB
High Global GM	384 MB
OS	Ubuntu 14.04
Kernel	3.18.0-rc7
Linux/ Windows Guest VM Configuration	
vCPU	2
Memory	1800 MB/ 2048MB
Low Global GM	64 MB/ 128 MB
High Global GM	384 MB
OS	Ubuntu 14.04/ Windows 7

Table 1: Experimental Configuration

Configurations All the VMs in this paper are run on one server configured as Table 1, and gScale is applied on gVirt's 2015Q3 release as a patch. To support higher resolution, fence registers have to serve larger graphics memory range. In our test environment, gScale reserves 300MB low global graphics memory size to be the fence

memory space pool, and this is enough for 15 VMs under the 1920*1080 resolution.

Benchmarks We mainly focus on the 3D workloads, for that in cloud environment graphics processing is still the typical GPU workload. Some 2D workloads are covered too. However, we only use 2D workloads to prove the full functionality of vGPUs hosted by gScale, because 2D workloads can also be accelerated by CPU. For Linux 3D performance, we choose the Phoronix Test Suit 3D marks[3], including Lightsmark, Nexuiz, Openarena, Urbanterror, and Warsow. Cairo-perf-trace[4] which contains a group of test cases is picked to evaluate Linux 2D performance. For Windows, we use 3DMark06[5] to evaluate 3D performance. PassMark[6] is chosen to evaluate 2D functionality. All the benchmarks are run under the 1920*1080 resolution.

Methodology We implemented a test framework that dispatches tasks to each VM. When all the tasks are completed, we collected the test results for analysis. When gScale hosts a large amount of VMs, I/O could be a bottleneck. We installed 3 SSD drives in our server and distributed the virtual disks of VMs in these SSD drives to meet VM's I/O requirement. For 3DMark06, the loading process takes a great amount of time, which leads to an unacceptable inaccuracy when run in mutiple VMs. Moreover, VMs start loading at the same time, but they cannot process rendering tasks simultaneously due to the different loading speed. To reduce the inaccuracy caused by loading, we run the 3DMark06 benchmark by splitting it into single units and repeat each unit for 3 times. The single units in 3DMark06 are GT1-Return To Proxycon, GT2-Firefly Forest, HDR1-Canyon Flight and HDR2-Deep Freeze, and they are for SM2.0 and SM3.0/HDR performance.

5.2 Scalability

In this section, we present the experiments of name's scalability on Linux and Windows. Figure 10 shows the 2D and 3D performance of Linux VMs hosted by gScale, scaling from 1 to 15, and the results of all the tests are normalized to 1VM. All the 3D performance in this paper is measured by value of frame per second (FPS) given by benchmarks. For most of our test cases, there is a clear performance degradation when the number of VMs is over 1, due to the overhead from copying private shadow GTT entries. The maximal degradations of Lightsmark, Nexuiz, Openarena, and Warsow

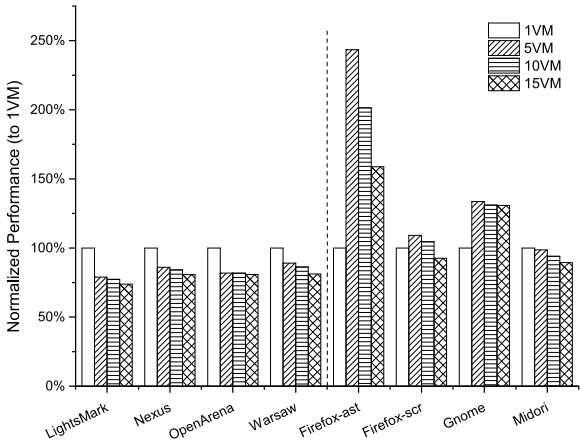


Figure 10: Scalability of gScale in Linux

are 26.2%, 19.2%, 19.3%, and 18.9%, respectively. For 3D workload Lightsmark, Nexuiz, Openarena, and Warsow, scaling from 5VM to 15VM, gScale achieves a negligible performance change. It demonstrates that GPU resource is efficiently shared among multiple VMs. For 2D workload, firefox-ast and gnome increase their performance from 1VM to 5VM, for that 2D workloads are also accelerated by CPU.

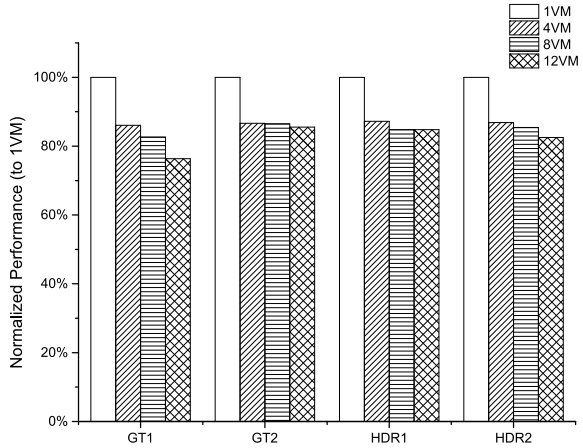


Figure 11: Scalability of gScale in Windows

The 3D performance of Windows VMs hosted by gScale scaling from 1 to 12 is in Figure 11, and all the test results are normalized to 1 VM. Similar with Linux, there is a visible performance degradation for each case when the number of VMs is over 1, and the maximal degradations of GT1, GT2, HDR1, and HDR2 are 23.6%, 14.5%, 15.2%, and 17.5%, respectively. The cause of degradation is the same with Linux VMs, which will be analyzed in Section 5.4. The performance scales well from 4VMs to 12VMs, and it proves that GPU resource is efficiently

[3]Phronix Test Suit, `http://www.phoronix-test-suite.com/`
[4]Cairo, `http://http://cairographics.org/`
[5]Cario, `http://www.futuremark.com`
[6]PassMark, `http://www.passmark.com`

utilized when the number of VMs increases.

5.3 Performance

Comparison with gVirt We compare the performance of gScale with gVirt in Figure 12, and the performance of gScale is normalized to gVirt. We examine the settings of 1-3 VMs for gScale, since gVirt can only support 3 guest vGPUs. For Linux, gScale achieves up to 99.89% performance of gVirt, while for Windows, gScale archives up to 98.58% performance of gVirt. There is a performance drop which is less than 5% of normalized performance when the number of instances is over 1. The performance decrease is due to copying the part of private shadow GTT for low graphics memory, and we will have a micro analysis in Section 5.4. This overhead is inevitable, for that global graphics memory space sharing will incur the overhead of copying private shadow GTT.

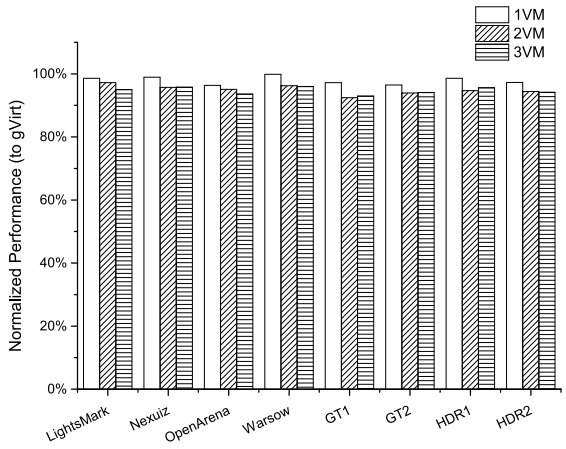


Figure 12: Performance Comparison

Performance Impact of Slot Sharing In this experiment, we want to evaluate the slot sharing of gScale under a high instance density. We launch 15 VMs at the same time. However, we only run GPU intensive workloads in some of them, while the rest VMs remain GPU idle. A GPU idle VM means a launched VM without GPU workload. We increase the number of GPU busy VM from 1 to 15, and observe the performance change. We use gScale-Basic to represent the gScale without slot sharing.

For 3D performance of gScale in Linux, we pick Nexuiz as a demonstration, and the case is run in an increasing number of VMs while gScale hosts 15 VMs in total, as shown in Figure 13. gScale and gScale-Basic has the same performance when the GPU busy VM is only one. When the number of GPU busy VMs increases, private shadow GTT copying happens. There is a 20% performance decrease for gScale-Basic. However, gScale has little performance degradation when the number of GPU busy VMs is less than 4, and slot sharing mitigates the performance degradation when the number of GPU busy VMs is less than 6. However, when the number of GPU busy VMs exceed 6, the slot sharing does not help with the overhead, and the performance is stable around 80% of normalized performance.

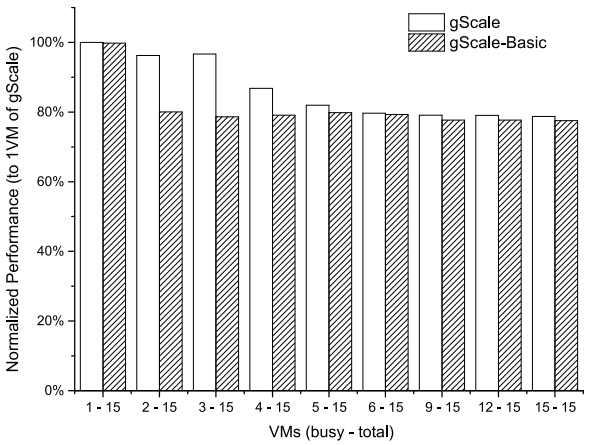


Figure 13: 3D Performance of Linux VMs

For 3D performance of gScale in Windows, GT1 is chosen to run in the rising number of VMs while gScale hosts 12 VMs in total. gScale shows the same performance with gScale-Basic when there is only 1 GPU busy VM. However, similar to the results on Linux, when the number of GPU busy VMs is over 1, there is a 16.5% performance degradation for gScale-Basic. gScale achieves a flat performance change when the number of GPU busy VMs is less than 4, and the results show that slot sharing mitigates the performance degradation before the number of GPU busy VMs reaches 6. When the number of GPU busy VMs exceed 6, the performance of gScale and gScale-Basic is very close.

5.4 Micro Analysis

Overhead of Private Shadow GTT We evaluate that overhead caused by copying private shadow GTT to show the performance optimization brought by slot sharing. Lightsmark and HDR2 are chosen to be the workloads in Linux and Windows VMs, respectively. We inspect the difference of overhead between gScale and gScale-Basic. For Linux, we launch 15 VMs, and run workloads in 3 of them. For Windows, run workloads in 3 VMs while total 12 VMs are launched. We measure the time of private shadow GTT copying and the time vGPU owns the physical engine in each schedule. Then, we collect the data from about 3000 schedules, and cal-

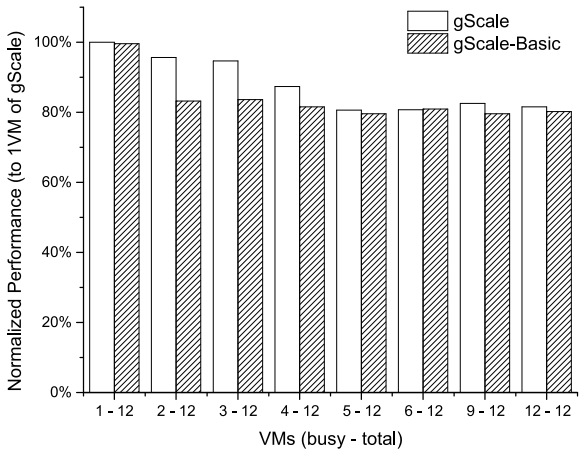


Figure 14: 3D Performance of Windows VMs

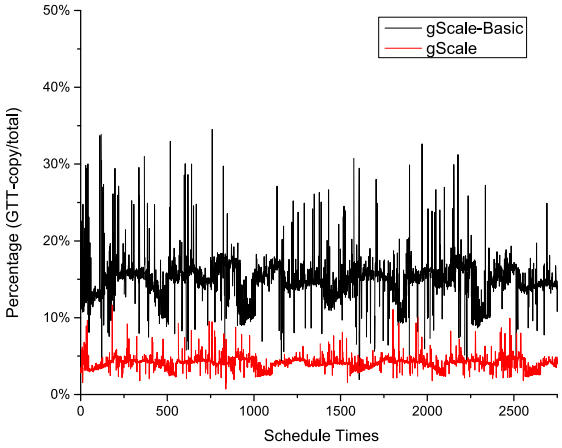


Figure 16: Overhead of Private Shadow GTT Copying in Windows

culate the percentage of how much time gScale takes to do the private shadow GTT copying in each schedule.

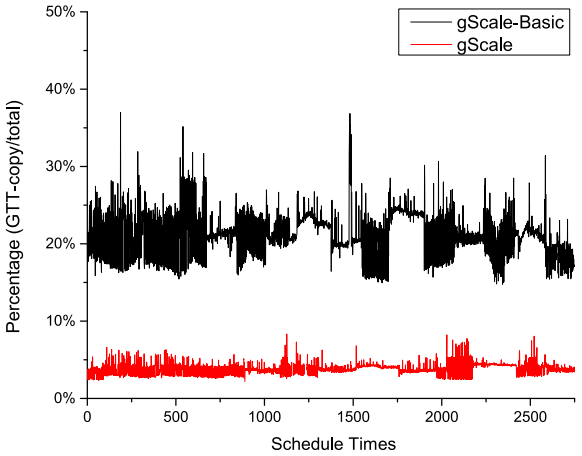


Figure 15: Overhead of Private Shadow GTT Copying in Linux

Figure 15 shows the overhead of gScale in Linux, for gScale-Basic (without slot sharing), the average overhead is 21.8%, while the average overhead of gScale is only 3.6%. In this case, slot sharing reduces the overhead of private shadow GTT copying by 83.4%. The overhead is dithering around the average value, for that shadow GTT copying needs memory bandwidth and CPU resource, which are also occupied by 3D workload.

Figure 16 shows the overhead of private shadow GTT copying in Windows, for gScale-Basic, the average overhead is 15.35%, while the average overhead of gScale is only 4.16%. In this case, slot sharing reduces the overhead of private shadow GTT copying by 72.9%. The slot sharing works better for Linux, because it only optimizes the overhead from high global graphics memory part of private shadow GTT copying, while we configure vGPU with twice the amount of low global graphics memory in Windows of that in Linux. Additionally, the overhead caused by the low graphics memory part of private shadow GTT copying is less than 5%, which is acceptable.

	LightsMark	Nexuiz	Openarena	Warsow	SM2.0	HDR
L Mapping(k)	18.8	4.67	4.9	6.6	10.2	8.1
GTT Modify(k)	455.3	313.5	228.5	1629.9	1134.2	1199.7
Percentage	4.13%	1.49%	2.14%	0.40%	0.90%	0.68%

Table 2: Frequency of Ladder Mapping

Frequency of Ladder Mapping Ladder mapping is constructed by gScale when CPU modifies the entry of GTT. We try to figure out the frequency of ladder mapping when 3D workloads are running. We count the total times of GTT modifications and the times of ladder mapping to calculate the percentage as shown in Table 2. For Windows workloads, the ladder mapping happens very rarely, which is less than 1%. For Linux, the percentage of ladder mapping frequency is higher than Windows, and we believe the reason is that the total amount of GTT modifications in Windows is a lot more than in Linux (up to 8x). At the same time, we observe a phenomenon that the ladder mapping mostly happens when workloads are being loaded, and it seldom happens when workloads are being processed. It explains the flat change of performance in our scalability evaluation, though ladder mapping could have overhead.

6 Discussion

Currently, gScale only supports Intel Graphics Processors. However, the principle of our design can be applied to other architectures. In addition to Intel, vendors like AMD, Qalcomm and Samsung also have integrated CPU/GPU systems and their graphics memory is also served by system memory [25]. Our ladder mapping could be applied to their solutions if they have similar requirements. Some GPUs, such as those from NVidia and AMD, may have dedicated graphics memory, but they also use graphics translation table to do address translation. We believe the concept of gScale's private shadow GTT could also help them share the graphics memory space. However, we could not test gScale on those GPUs, beacuse they are not open-source distributions.

7 Related Work

Using modern GPUs in a shared cloud environment remains challenge with a good balance among performance, features and sharing capability [30]. A lot of research efforts have been made to enable GPUs in virtual machines (i.e., Device emulation, API forwarding, Device Pass-through, and full GPU virtualization).

Device emulation is considered impractical because GPU hardware is vendor-specific and modern GPUs are complicated. Thus, QEMU [14] has emulated a legacy VGA device with a low performance to support only some basic functionality.

API forwarding has been widely studied and has been applied to many virtualization software already. By installing a graphics library in a guest OS, graphic commands can be then forwarded to the outside host OS. Host OS can execute those commands directly using the GPU hardware. WireGL [20] and Chromium [21] intercept OpenGL commands and parallelly render them on commodity clusters. VMGL [22] makes use of Chromium to render guest's OpenGL commands on the host side. GViM [19], rCUDA [18], and vCUDA [27] virtualize GPGPU applications by forwarding CUDA commands in virtual machines. Kernel consolidations have been studied for efficiency with Kernelet [32]. However, one major limitation of API forwarding is that the graphic stack on guest and host must match. Otherwise, host OS is not able to process guest's commands. For example, a Linux host cannot execute DirectX commands forwarded by a Windows guest. As a result, a translation layer must be built for Linux host to execute DirectX commands: Valve [8] and Wine [10] have built such translation layers, but only a subset of DirectX commands is supported; VMWare [17] and Virgil [9] implement a graphic driver to translate guests' commands to their own commands.

Device Pass-through achieves high performance in GPU virtualization. Recently, Amazon [2] and Aliyun [1] have provided GPU instances to customers for high performance computing. Graphic cards can be also passed to a virtual machine exclusively using Intel VT-d [13, 15]. However, direct pass-through GPU is dedicated, and also sacrifices the sharing capability.

Two full GPU virtualization solutions have been proposed, i.e., gVirt [30] and GPUvm [28, 29], respectively. GPUvm implements GPU virtualization for NVIDIA cards on Xen, which applies several optimization techniques to reduce overhead. However, full-virtualization will still cause non-trivial overhead because of MMIO operations. A para-virtualization is also proposed to improve performance. Furthermore, GPUvm can only support 8 VMs in their experimental setup. gVirt is the first open source product level full GPU virtualization solution in Intel platforms. It provides each VM a virtual full fledged GPU and can achieve almost native speed. Recently, gHyvi [16] uses a hybrid shadow page table to improve gVirt's performance for memory-intensive workloads. However, gHyvi inherits the resource partition limitation of gVirt, so it also suffers from the scalability issue too.

NVIDIA GRID [7] is a commercial GPU virtualization product, which supports up to 16 VMs per GPU card now. AMD has announced its hardware-based GPU virtualization solution recently. AMD multiuser GPU [3], which is based on SR-IOV, can support up to 15 VMs per GPU. However, neither NVIDIA nor AMD provides public information on technical details.

8 Conclusion and Future Work

gScale addresses the scalability issue of gVirt with a novel sharing scheme. gScale proposes the *private shadow GTT* for each vGPU instance, which allows vGPU instances to share the part of global graphics memory space only visible to GPU. A *ladder mapping* mechanism is introduced to make CPU directly access host physical memory space serving the graphics memory without referring to global graphics memory space. At the same time, fence memory space pool is reserved from low graphics memory space to ensure the functionality of fence registers. gScale also implements *slot sharing* to improve the performance of vGPU under a high instance density. Evaluation shows that gScale scales well up to 15 vGPU instances in Linux or 12 vGPU instances in Windows, which is 5x and 4x scalability compared to gVirt. Moreover, gScale archives up to 96% performance of gVirt under a high density of instances.

As for future work, we will focus on optimizing the performance of gScale, especially when gScale hosts large amount of instances with intensive workloads. To

exploit the performance improvement of slot sharing, we will design a dynamic deploy policy based on the workload of instances.

9 Acknowledgements

We acknowledge our shepherd Nisha Talagala and the anonymous reviewers for their insightful comments. This work was supported by National Infrastructure Development Program (No. 2013FY111900) NRF Singapore CREATE Program E2S2, STCSM International Cooperation Program (No. 14510722600), and Shanghai Key Laboratory of Scalable Computing and Systems.

References

[1] Alihpc. https://hpc.aliyun.com/product/gpu_bare_metal/.

[2] Amazone high performance computing cloud using gpu. http://aws.amazon.com/hpc/.

[3] Amd multiuser gpu. http://www.amd.com/en-us/solutions/professional/virtualization.

[4] Intel graphics driver. http://www.x.org/wiki/IntelGraphicsDriver/.

[5] Intel graphics virtualization technology (intel gvt). https://01.org/zh/igvt-g.

[6] Intel open source hd graphics programmer's reference manual (prm). https://01.org/linuxgraphics/documentation/hardware-specification-prms.

[7] Nvidia grid: Graphics-accelerated virtualization. http://www.nvidia.com/object/grid-technology.html.

[8] Valve togl. https://github.com/ValveSoftware/ToGL.

[9] Virgil 3d gpu project. https://virgil3d.github.io.

[10] Wine project. http://winehq.org/.

[11] Xen project release features. http://wiki.xenproject.org/wiki/Xen_Project_Release_Features.

[12] Xengt setup guide. https://github.com/01org/XenGT-Preview-kernel/blob/master/XenGT_Setup_Guide.pdf.

[13] ABRAMSON, D. Intel virtualization technology for directed i/o. *Intel technology journal 10*, 3 (2006), 179–192.

[14] BELLARD, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[15] DONG, Y., DAI, J., HUANG, Z., GUAN, H., TIAN, K., AND JIANG, Y. Towards high-quality i/o virtualization. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference* (2009), ACM, p. 12.

[16] DONG, Y., XUE, M., ZHENG, X., WANG, J., QI, Z., AND GUAN, H. Boosting gpu virtualization performance with hybrid shadow page tables. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 517–528.

[17] DOWTY, M., AND SUGERMAN, J. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[18] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTÍ, E. S. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on* (2010), IEEE, pp. 224–231.

[19] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing* (2009), ACM, pp. 17–24.

[20] HUMPHREYS, G., ELDRIDGE, M., BUCK, I., STOLL, G., EVERETT, M., AND HANRAHAN, P. Wiregl: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 129–140.

[21] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In *ACM Transactions on Graphics (TOG)* (2002), vol. 21, ACM, pp. 693–702.

[22] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. Vmm-independent graphics acceleration. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), ACM, pp. 33–43.

[23] NEIGER, G., SANTONI, A., LEUNG, F., RODGERS, D., AND UHLIG, R. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal 10*, 3 (2006).

[24] PHULL, R., LI, C.-H., RAO, K., CADAMBI, H., AND CHAKRADHAR, S. Interference-driven resource management for gpu-based heterogeneous clusters. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* (2012), ACM, pp. 109–120.

[25] PICHAI, B., HSU, L., AND BHATTACHARJEE, A. Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces. *ACM SIGPLAN Notices 49*, 4 (2014), 743–758.

[26] QI, Z., YAO, J., ZHANG, C., YU, M., YANG, Z., AND GUAN, H. Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming. *ACM Transactions on Architecture and Code Optimization (TACO) 11*, 2 (2014), 17.

[27] SHI, L., CHEN, H., SUN, J., AND LI, K. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on 61*, 6 (2012), 804–816.

[28] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: why not virtualizing gpus at the hypervisor? In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference* (2014), USENIX Association, pp. 109–120.

[29] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: Gpu virtualization at the hypervisor. *Computers, IEEE Transactions on PP*, 99 (2015), 1–1.

[30] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A full gpu virtualization solution with mediated pass-through. In *Proc. USENIX ATC* (2014).

[31] WALDSPURGER, C. A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36*, SI (2002), 181–194.

[32] ZHONG, J., AND HE, B. Kernelet: High-throughput gpu kernel executions with dynamic slicing and scheduling. *IEEE Trans. Parallel Distrib. Syst. 25*, 6 (June 2014), 1522–1532.

A General Persistent Code Caching Framework for Dynamic Binary Translation (DBT)

Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant
University of Minnesota, Twin Cities
{wenwang, yew, zhai, mccamant}@cs.umn.edu

Abstract

Dynamic binary translation (DBT) translates binary code from one instruction set architecture (ISA) to another (same or different) ISA at runtime, which makes it very useful in many applications such as system virtualization, whole program analysis, system debugging, and system security. Many techniques have been proposed to improve the efficiency of DBT systems for long-running and loop-intensive applications. However, for applications with short running time or long-running but with few hot code regions such as JavaScript and C# applications in web services, such techniques have difficulty in amortizing the overhead incurred during binary translation.

To reduce the translation overhead for such applications, this paper presents a general persistent code caching framework, which allows the reuse of translated binary code across different executions for the same or different applications. Compared to existing approaches, the proposed approach can seamlessly handle even dynamically generated code, which is very popular in script applications today. A prototype of the proposed framework has been implemented in an existing retargetable DBT system. Experimental results on a list of applications, including C/C++ and JavaScript, demonstrate that it can achieve 76.4% performance improvement on average compared to the original DBT system without helper threads for dynamic binary translation, and 9% performance improvement on average over the same DBT system with helper threads when code reuse is combined with help threads.

1 Introduction

Dynamic binary translation or transformation (DBT) has been widely used in many applications such as system virtualization, whole program analysis, system debugging, and system security. Most notable examples include QEMU [6], DynamoRIO [4], Pin [17], Valgrind [18], and many others [9, 25, 5]. DBT systems can dynamically translate *guest* binary code in one instruction set architecture (ISA) to another *host* ISA (same as or different from guest ISA), and achieve the emulation of guest applications on the host machines or the enhanced functionalities of the guest binaries. It bypasses the need of an intermediate representation such as bytecode in language-level virtual machines, e.g. Java virtual machine (JVM) or Dalvik in Android.

Compare to native execution, DBT systems usually consist of two phases. In the first phase, guest binaries are emulated and profiled on the host system to detect the ”hotness” of code regions. Hot code regions are then translated and stored in a code cache. It allows the execution to enter the second phase in which the translated binaries in the code cache are executed without further code emulation or translation. Typically, for long-running and loop-intensive guest applications, 90% of the execution time could stay in the second phase [21]. It allows the overhead incurred in the first phase to be substantially amortized.

Many techniques have been proposed to improve the efficiency of the DBT systems and the performance of the translated host binaries [26, 27, 10]. Nevertheless, for many applications with short running time or long-running applications with few hot code regions, e.g., JavaScript and C# applications in web services [11] that require fast response time and high throughput, such techniques have difficulty in amortizing the overhead from the first phase.

Figure 1 shows the translation overhead (incurred in the first phase) of SPEC CINT2006 using HQEMU [13], a QEMU-based retargetable DBT that can dynamically translate guest binaries across several different major ISAs such as x86-32, x86-64 and ARM. To more accurately measure the translation overhead of short-running applications, the small *test* input is used to shorten their running time.

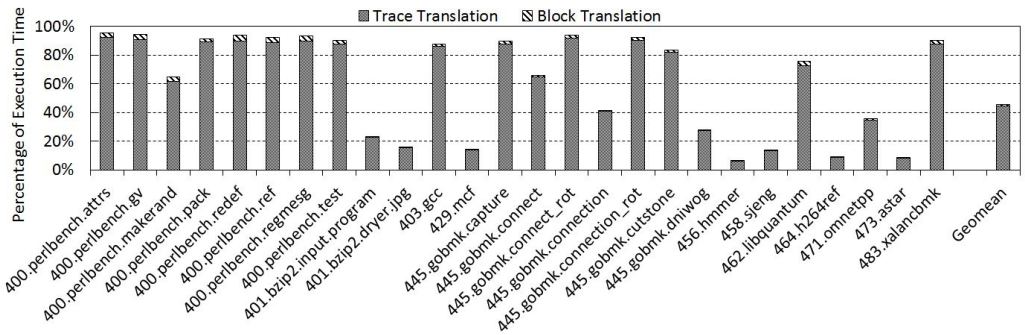


Figure 1: Translation overhead in HQEMU for SPEC CINT2006 with *test* input.

As shown in Figure 1, more than 40% of the execution time on average is spent in the first phase for dynamic translation (more details are discussed in Section 2). Therefore, reducing the translation overhead can significantly improve the performance of these short-running applications or those with few hot code regions. Additionally, lower translation overhead can help to reduce power consumption - a benefit critical to mobile devices with limited battery life. This is one of the reasons Google switches Android runtime from Dalvik to ART with native binaries [1].

One possible approach to reduce translation overhead is to use *static* binary translation (SBT), and perform binary translation offline. It can completely avoid such translation overhead at runtime. However, SBT has many open issues to deal with, such as code discovery for ISAs with variable-length instructions (e.g., Intel x86) and indirect branches with unknown branch targets [8, 24]. Also, it is difficult for SBT to leverage runtime information to optimize the performance of the translated code.

Besides SBT, helper threads have been proposed to shift the translation overheads to other threads [13]. Even though the translation overheads can be hidden substantially this way due to concurrency, it can reduce the system throughput, which could be sensitive in a cloud environment. To reduce re-translation overhead caused by code cache invalidation, annotation-based and inference-based schemes have been proposed to mitigate such overheads [11]. Even so, they still require re-translation the first time those guest binaries are re-encountered.

Compared to those approaches, persistent code caching is an effective alternative to reduce translation overhead [12, 7, 21, 22]. In this approach, the translated binaries are reused the next time the same guest application is executed (i.e., the generated binaries persist across different runs of the same application), or reused by other applications (i.e., the generated binaries such as those in shared libraries persist across different runs of different applications). The translation overhead can thus be reduced (or eliminated). Existing persistent code caching approaches [7, 21, 22] leverage guest instruction address or offset to detect persistent code hits when re-using persistent code. It limits their applicability to unmodified guest binary code at the same instruction address or offset across executions. However, different executions are very likely to have different guest binary code at the same instruction address or offset in practice, e.g., dynamically generated guest binary code.

In this paper, we focus on some main challenges in DBT systems using persistent code caching. The first is the need to deal with relocatable *guest* binaries. The second is the need to generate relocatable *host* binaries in order to persist across different runs for reuse. The third is the need to deal with the dynamically generated code by the guest applications, e.g., if the guest binary is dynamically generated by a just-in-time (JIT) engine, we need to deal with both the dynamically generated guest binaries by the JIT engine and the JIT engine itself [11], so both can persist across different runs.

Relocatable Guest Binaries. Typically, there are two kinds of relocatable guest binaries, (1) position-independent guest binaries, and (2) guest binaries that contain relocatable meta-data for load-time relocation. If a guest binary is position-independent, the offset from its starting address can be used to index and search for its persistent code [7]. However, a shared library may have different offsets when statically linked in different applications, hence, the offset alone is not sufficient to reuse persistent code across statically linked applications that share the same libraries. For the guest binaries that contain relocatable meta-data for load-time relocation, it is also not sufficient to index and search for its persistent code with only addresses of the guest binary code [21].

Relocatable Host Binaries. Position-dependent addresses exist in the translated host instructions need to be relocated. A typical example is the translation of a guest *call* instruction into two host instructions, *push* `next_guest_PC` and *jmp*. The position-dependent

branch target `next_guest_PC` needs to be made position independent (i.e., relocatable) in order to be effectively reused across different execution runs.

Another example is the exit stub in a translated code trace for returning to the DBT runtime system when the next guest code block is untranslated. The returned address of the next guest block needs to be made relocatable as this address could be different in different execution runs due to relocatable guest binaries.

In addition, a position-dependent host address could be embedded in a translated host instruction to jump back from the translated host code to the translator. A possible solution to this challenge is to ask the DBT to generate position-independent code and to avoid using host instructions with embedded position-dependent addresses. However, this kind of implementation is not always efficient [7]. Especially for applications with long running time or mostly covered with hot code regions that do not require persistent code caching, position-independent code could introduce unnecessary runtime overhead.

Dynamically Generated Guest Binaries. JIT engines have been widely adopted in many languages such as Java, JavaScript, and C#. Persistent code caching for dynamically generated code by such JIT engines is very challenging because their addresses are very likely to change across runs due to address space layer randomization (ASLR) used by operating systems for security reasons. Furthermore, despite the availability of bytecode for such applications, their dynamically generated binaries are often linked with other application-specific libraries making them difficult to port across different ISAs with only bytecode. One study shows that an average of more than 50% of such codes are in native binaries [15]. Many of the applications written in languages such as JavaScript also either have very short running time or have very few hot code regions [20]. Persistent code caching is an effective way to reduce re-translation overhead and to improve performance when ported across different ISAs.

This paper aims at addressing the above challenges in persistent code caching for DBT systems across different ISAs. We generate and maintain relocation records in the persistent code to get around the issues of position-dependent code mentioned above. Using such relocation records, host instructions that contain position-dependent addresses can be relocated before it is used. We also keep the original guest binary code in the generated persistent code. Only the persistent code that have the matched guest binary is reused. In this way, persistent code for dynamically generated code can be reused seamlessly even if the address of the generated code is changed in a later execution run.

A prototype of such a persistent code caching framework has been implemented in an existing retargetable DBT system. Experimental results on a set of benchmarks that include C/C++ and JavaScript applications demonstrate the effectiveness and efficiency of our approach. Without trying to optimize our prototype, 76.4% and 9% performance improvements on average are observed compared to the original system without and with helper threads for dynamic binary translation, respectively. Experiments on using persistent code across different execution runs for the same application with different inputs show that the size of the persistent code traces affect the benefits of such persistent code. Additionally, experiments on the persistent code accumulation across different applications suggest that an effective persistent code management is required.

The contributions of this paper are as follows:

- A general persistent code caching framework using relocation records and guest binary code matching is proposed to allow position-dependent host binaries to be reused across different execution runs. This approach is also applicable to dynamically generated guest binaries from JIT compilers. To the best of our knowledge, this is the first attempt to allow dynamically generated guest code to persist across different execution runs.
- A prototype for such a persistent code caching framework has been implemented in an existing DBT system, HQEMU [13], which is a QEMU-based retargetable DBT system.
- A number of experiments are conducted to demonstrate the effectiveness and efficiency of this approach. 76.4% and 9% performance improvements are observed with the help of persistent code caching compared to a retargetable DBT system (i.e., HQEMU) without and with the helper threads for dynamic binary translation.

The rest of the paper is organized as follows. Section 2 describes the proposed persistent code caching approach. Section 3 discusses some important implementation issues. Section 4 presents our experimental results. Section 5 discusses other related work. And Section 6 concludes our paper.

2 Persistent Code Caching - Our Approach

In general, DBT systems translate guest binary into host binary at the granularity of a *basic block* (or *block* for short), which is a sequence of guest instructions with only one entry and one exit. The generated host code

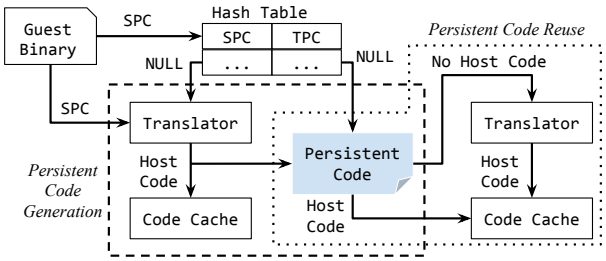


Figure 2: Persistent code caching for DBT systems.

is stored in an executable memory region, called *code cache*. The measured translation overhead in SPEC CINT2006 using *test* input is shown as *Block Translation* in Figure 1. After the translation of a guest block, the DBT system then transfers the execution to the generated host code in the code cache. During the execution in the code cache, a jump back to the translator/emulator is triggered when an untranslated block is encountered, and the translation is restarted.

In a DBT system, a *hash table* is used to manage the mapping from the *guest* program counter (PC) of the source binary, i.e., SPC, to the corresponding *host* PC of the translated code, i.e., TPC. Each time a new block is translated, the hash table is updated with the pair of SPC and TPC. Indirect branches and return instructions can look up the hash table to find out the TPC of the target block. A *jump stub* is used to chain translated code of two blocks connected via direct branch instruction to avoid the jump back to the translator, which is extremely time-consuming. The jump stub is initialized to jump back to the translator at first. After target block is translated, it is patched to jump to TPC of target block.

A hot path in the guest binary forms a *trace*, which contains several blocks with one entry and multiple exits. The number of blocks in a trace is the *size* of the trace. After a trace is formed, the blocks in this trace are re-translated and more aggressive optimizations can be applied to achieve a better performance. The translation/optimization overhead in this process for the SPEC CINT2006 benchmarks is shown as *Trace Translation* in Figure 1.

In our persistent code caching approach, translated host code for both basic blocks and traces are kept and reused across executions, i.e., they persist across executions. Thus, both block translation and trace translation overheads can be reduced. Figure 2 shows the framework and the work flow of our persistent code caching approach. It consists of two phases, *persistent code generation* and *persistent code reuse*.

In our approach, persistent code is organized in *entries*. As shown in Figure 2, after the translator finishes the translation of a basic block or trace, the generated host binary code and the related information such as guest binary code, relocation information and internal data structures are copied to a host memory region and form a new *persistent code entry*. The host memory region is called *persistent code memory* and is allocated at the start of the DBT system. At the end of the program execution, e.g., a guest `exit_group` Linux system call is emulated, all entries in the persistent code memory are flushed to the disk and stored in a *persistent code file*, which is used across different executions.

To reuse the persistent code generated in previous executions, the persistent code file is loaded into the memory at the start of the DBT system and installed into a two-level hash table, called *PHash* (persistent code hash table). The details of PHash is described in Section 2.2.

Before a guest block or trace is translated, PHash is looked up to check if there is a matching persistent code entry already. Here, *matching* means they have the same guest binary code. If a matched persistent code entry is found, the translator is bypassed and the translated host binary in this entry is copied to the code cache directly. Before the execution of the copied host binary, the required relocation and recovery are performed to ensure its correct execution. In this way, the host binary generated in one execution can be correctly reused in another execution, and the translation overhead is reduced or eliminated.

In addition, our persistent code caching approach supports *persistent code accumulation* across different executions. To accumulate persistent code, those basic blocks and traces that cannot find their matching entries in PHash will form new persistent code entries in persistent code memory. At the end of the current execution, these new persistent code entries are merged with existing persistent code entries to produce a new persistent code file.

Our persistent code caching approach can be integrated into most of the existing DBT systems. Only small changes to the translation work flow is required to generate, reuse, and accumulate persistent code. In the following sections, we describe in more details about our approach.

2.1 Persistent Code Generation

We form a new persistent code entry each time a basic block or a trace is translated. Each formed persistent code entry is comprised of three parts: *meta-data for the host binary code* (MDHBC), *guest binary code* (GBC), and *host binary code* (HBC). The organization of these three parts is shown in Figure 3.

MDHBC has two kinds of information: internal data structure and relocation information. The internal data structure is used to recover its corresponding

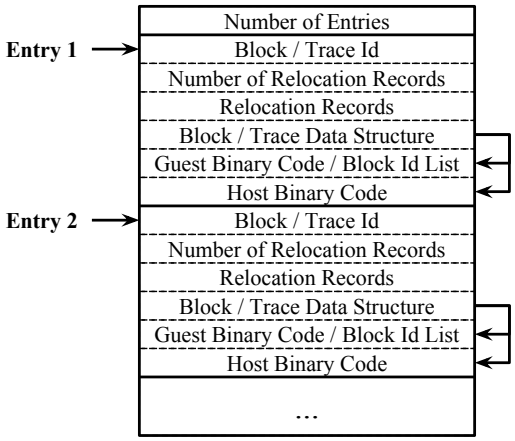


Figure 3: Detailed structure of persistent code file.

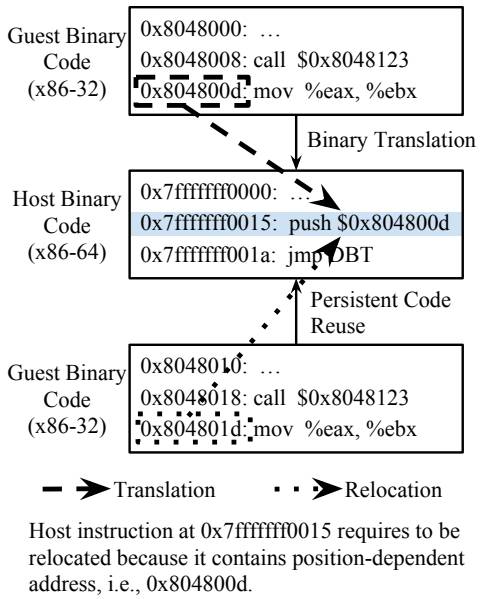


Figure 4: Host instruction relocation.

data structure to ensure the correct execution of the reused host binary code during the execution when the persistent code is used. The relocation information is used to convert the position-dependent addresses embedded in the host instructions. GBC is used to find and verify the matching persistent code entry. And HBC is the main part of persistent code reused across executions.

Meta Data for Host Binary Code (MDHBC). To generate MDHBC, we make a copy of the internal data structure of a block/trace when its translation is completed. Here, the internal data structure can vary between different DBT systems. In our prototype system, it is the data structure used by the DBT system (i.e., HQEMU) to represent each translated block/trace. A unique id is assigned to each copied data structure, called *block/trace id*. Typically, one of the most accessed items in this data structure is the offset of jump stubs in translated host binary code, which is used by the DBT system to chain two translated blocks/traces.

Another important part of MDHBC is the relocation information. For each block/trace, the relocation information is organized into records. Each relocation record is created for a host instruction whose operands contain a position-dependent address. Note, if a host instruction contains a guest position-dependent address and this address is also contained in the translated guest instruction, we do not create relocation record for such a host instruction because our guest binary code matching mechanism can filter out blocks/traces that include this guest instruction correctly if this position-dependent address is changed across executions. We use the example shown in Figure 4 to explain the information required for relocating host instructions.

In this example, the guest instruction *call* at guest address `0x8048008` is translated into two host instructions *push* `0x804800d` and *jmp* at host addresses `0x7fffffff0015` and `0x7fffffff001a`, respectively. If the translated code is reused across executions, the host instruction *push* `0x804800d` is required to be relocated because it contains guest position-dependent address `0x804800d`, which is the return address of the *call* instruction, or the address of the guest instruction following the call instruction. To relocate such a host instruction, two kinds of information are required. (1) *The location of the position-dependent address in code cache*, which is usually the start address of the relocated host instruction plus the size of its opcode. In this example, the location of `0x804800d` in code cache is `0x7fffffff0015` plus the size of *push* opcode, which is `1`. (2) *The correct address that should be put into this location*. In this example, the host code is reused by the basic block at address `0x8048010` with a *call* instruction at `0x8048018`. Thus, the correct address is `0x804801d`, which is the return address of this *call* instruction.

Each relocation record saves two items to keep the two required pieces of information. The first is the *offset* from the starting address of the translated host code for this block/trace to the location of the position-dependent address. In the above example, the offset is `0x16`. After the host code is copied from a persistent code entry to the code cache for reuse, this offset plus the starting address of the copied host code is the address for relocation. The second is a *source id* that identifies the source of the correct address. We use an enumerate type for the source id, called `IDType`. In the above example, the source id is `GUEST_NEXT_PC`, which means the correct address is

the address of the guest instruction following the last instruction in this block.

To facilitate the generation of relocation records, a unified API is provided for DBT developers:

```
extern int gen_reloc_record (int    offset,
                             IDType source,
                             BLOCK  *block,
                             TRACE  *trace);
```

This function generates a relocation record for `block` or `trace` in its MDHBC. Specifically, it should be called when a position-dependent address is inserted into a host instruction during binary translation.

Guest Binary Code. A copy of the guest binary code of a block is placed in the persistent code memory after its translation. The address information of this copy is added to the data structure of this block in MDHBC for future access. To limit the size of the generated persistent code, instead of copying the guest binary code of blocks in a trace, we only keep a list of block ids in the copied data structure of this trace in MDHBC.

Host Binary Code. After the translation of a block/trace, the translated host code is copied from code cache to the persistent code memory. Also, the address information of this copy needs to be saved into the data structure of the block/trace in MDHBC.

At the end of the execution, all formed entries in persistent code memory are flushed to disk to produce a persistent code file. The number of persistent code entries is stored at the beginning of the file, followed by all entries, as illustrated in Figure 3.

2.2 Persistent Code Reuse

After the generation of persistent code, it can be reused across executions. This section describes how to reuse persistent code.

At the beginning of an execution, the persistent code file is loaded from disk into memory. Each entry in the file is processed and installed into a two-level hash table, i.e., *PHash*. Figure 5 shows the structure of PHash (part of the second level hash tables is omitted due to space limitation). Relocation and data structure recovery will be done later after a matching persistent code entry is found for a block/trace.

We use L1-PHash and L2-PHash to represent the first and the second level hash tables in PHash. The hash keys of L1-PHash and L2-PHash are generated based on the guest binary code in each persistent code entry. If an entry corresponds to a trace, the guest binary of its first basic block is used to generate the hash key. Note, the property of the translated host binary code cannot be used as

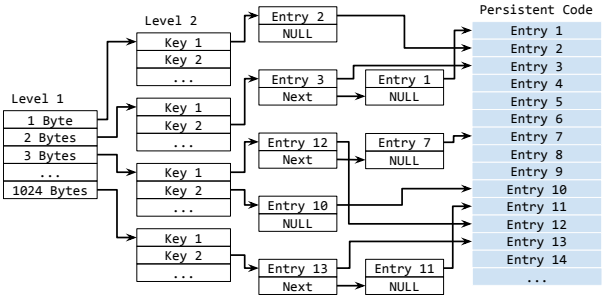


Figure 5: Two-level hash table used to look up persistent code for reuse.

the hash keys, because there is no translated host binary code but the original guest binary code of a block/trace when PHash is looked up to find out the matched entry for this block/trace.

Specifically, the hash key of L1-PHash is generated using the size of the guest binary code in bytes. The reason is that guest binary with very large basic blocks is very rare in practice, which can help to limit the size of L1-PHash. In our experiments, we found 1024 is enough for all guest applications evaluated. At the second level, several bytes in the guest binary are chosen to construct a byte combination. With this byte combination as input, a simple hash function is used to generate an integer in the range from 1 to 256, which is used as the hash key of L2-PHash. Compared to a single-level hash table, this two-level hash table can achieve comparable performance with a smaller overall table size for persistent code with a very large number of entries.

To limit the size of PHash, L2-PHash is allocated only when at least one persistent code entry hit it. In other words, L1-PHash is allocated first and initialized with `NULL`. When an entry hits a `NULL` in L1-PHash, a new L2-PHash is allocated and its address is used to update the corresponding `NULL` entry in L1-PHash. After filling all persistent code entries, PHash can be used in this execution without further update.

To find a matching persistent code entry for a basic block, its guest binary is used to look up PHash. The boundary of this basic block, i.e., the start and the end addresses of this basic block, should be identified first, The start address is usually available in existing DBT systems. However, the end address is unavailable until the basic block is being decoded, and a branch instruction is encountered or the maximum number of guest instructions for a basic block is reached [6, 13]. After this, the guest binary code of this basic block can be used to look up PHash. Algorithm 1 describes the details of such a lookup for a basic block.

First, the size of the guest binary code of this block is used to look up L1-PHash. Second, an hash key is

Algorithm 1: Persistent Code Reuse

Input: A basic block *bb*, with the start address and size of the guest binary code, *addr* and *size*
Output: Persistent code entry matching with *bb*, otherwise NULL

```
1  h ← PHash [size];
2  if h = NULL then
3      return NULL;
4  key ← hash_func (addr, size);
5  p ← h [key];
6  while p ≠ NULL do
7      if guest_binary_same (p, addr, size) then
8          do_host_code_copy (p, bb);
9          do_host_code_relocation (p, bb);
10         do_data_structure_recovery (p, bb);
11         return p;
12     p ← p → next;
13 return NULL;
```

calculated based on the guest binary code of this block using a simple hash function to look up L2-PHash. Last, the list pointed by the hit entry in L2-PHash is scanned. The guest binary code in each persistent code entry on this list is compared with the guest binary code of the input block until a matching entry is found. If there is no such matching entry, NULL is returned and the translator is re-invoked to perform the translation. The look-up procedure for a trace is similar to a basic block. The only difference is that the guest binary code of all basic blocks in the trace is compared to find out a matching entry.

After a matching persistent code entry is found, the host code in this entry is copied to code cache. Then the relocation records in this entry is applied to the copied host code. Note, the copied code is still unchained at this point. However, after we recover the internal data structure, block/trace chaining will be done automatically by the translator during the ensuing execution.

Global optimizations applied by DBT systems might complicate the above look-up process. For example, some global optimizations for the current basic block might be based on its predecessor and successor blocks. A typical example is the conditional code optimization for Intel x86-like ISAs that have side effects on condition codes in some instructions [19]. It can eliminate unnecessary conditional code generation in current block based on the definition and usage of conditional codes in its predecessor and successor blocks. In such cases, if the guest binary code of the predecessor or successor blocks is changed, the translated host binary code of current block cannot be reused.

To deal with this issue, the guest binary of all blocks that may affect the translation of the current block should be compared to ensure the consistency of its translated host code. This can be realized by two steps. First, in persistent code generation phase, blocks that affect the translation decisions of current block are saved to MD-HBC when a global optimization is applied. Second, during the above lookup process, the guest binary of *all* saved basic blocks, not just the current block, are compared to determine a matching persistent code entry for the current block.

2.3 Persistent Code Accumulation

Even with persistent code, a basic block or a trace may still need to be translated because it may not have a matching persistent code entry found. In our approach, these newly translated host code is accumulated for future reuse. To accumulate persistent code from multiple execution runs, we form new persistent code entries for newly translated blocks and traces in each execution. At the end of the execution, the newly formed persistent code entries are merged with existing entries to generate a new persistent code file. Section 4 discusses the effectiveness of persistent code accumulation across same and different guest applications.

3 Implementation

A prototype of our persistent code caching approach has been implemented in an existing retargetable DBT system, HQEMU [13], which is a retargetable DBT system based on QEMU [6]. HQMEU uses original translator in QEMU, i.e., Tiny Code Generator (TCG), to translate basic blocks. Moreover, LLVM JIT [14] is used to generate more efficient code for traces. To detect traces, HQEMU inserts a *profile stub* and a *predict stub* into host binary generated by TCG for each basic block. The profile stub is used to count the number of dynamic executions of this basic block. Once the counter reaches a preset threshold, the profile stub is disabled and the *predict stub* is enabled to form the trace.

After a trace is formed, HQEMU converts TCG intermediate representations (IR) of the basic blocks in this trace to LLVM IR. Then, an LLVM JIT engine takes the LLVM IR as input and generates host binary code on the fly. During this process, several LLVM optimization passes are applied to improve the efficiency of the generated host code. Additional translation/optimization overhead is also introduced due to the time-consuming LLVM optimizations applied. To mitigate such translation/optimization overhead, HQEMU spawns helper threads to perform translations/optimizations for traces. However, this solution can reduce the system throughput, which could be sensitive in a cloud environment.

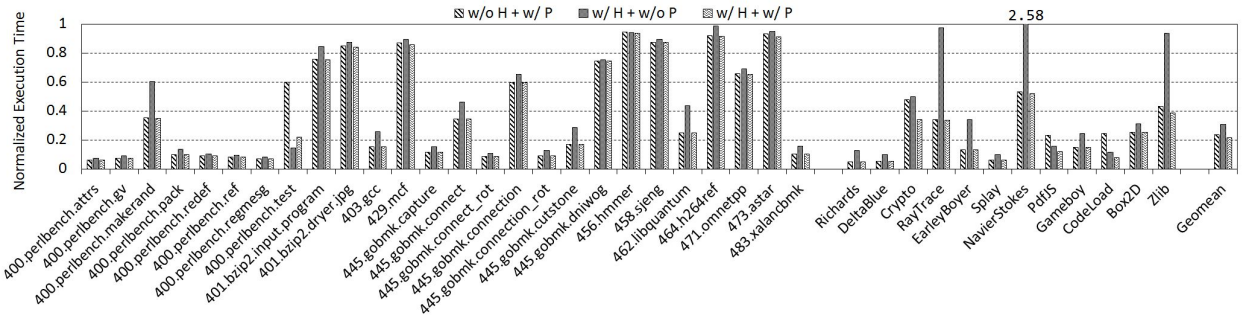


Figure 6: Performance improvement achieved by our persistent code caching approach. The base line is the execution without helper thread and persistent code, i.e., `w/o H + w/o P`.

The first issue that needs to be addressed in our implementation is *helper function calls*. For retargetability purpose, HQEMU (also in QEMU) leverages helper functions in the translator to emulate some complicated guest instructions, e.g., floating point and SSE instructions. These helper functions are called directly from code cache via position-dependent addresses. However, due to ASLR used by operating systems for security reasons, these function addresses are very likely to change across executions. Therefore, these function calls needs to be relocated for persistent code caching purpose. In our implementation, we use the unified API presented in Section 2.1 to generate relocation entries for helper function calls generated by TCG. For helper function calls generated by LLVM JIT, to simplify the implementation, we leverage redundant debugging meta data to acquire the relocation information from LLVM JIT.

Another special issue is about *global memory* and *constant memory pool* allocate by LLVM JIT during translating/optimizing traces. Global memory is a memory region used to save global variables that can be accessed by different traces. Constant memory pool is a local memory region allocated by LLVM JIT only for some specific traces, which is followed by the translated host code for these traces. For global memory, we save the initial value of each global variable into MDHBC once it is allocated by LLVM JIT. These values are used to recover the global memory in future executions. For constant memory pool, we adjust the starting address of translated host code for each trace to start from the beginning of the memory pool, if it exists. Thus, the constant memory can be saved and recovered along with the host code of traces.

Here, we focus on x86-32 to x86-64 cross-ISA translation. In the implementation, the profile stub and the predict stub are enabled and disabled as in original HQEMU to detect and form traces, whether or not the persistent code is available. Besides, each jump stub in host binary code copied from persistent code entries is set to jump back to the translator at first and patched later by the translator via block/trace chaining automatically. To guarantee the atomicity of the patch operation, which is a write to a 32 bit memory location, the address of the location should be 4-byte aligned on the host platform. Fortunately, HQEMU allocates code cache for each block/trace from an aligned address, which solves this issue naturally.

4 Experimental Study

Two types of guest applications, SPEC CINT2006 and SpiderMonkey [3] are employed to evaluate the performance. For SPEC CINT2006, the complete suite is included using *test* input. SpiderMonkey is a JavaScript engine written in C/C++ from Mozilla, which has been deployed in various Mozilla productions, including Firefox. The performance of SpiderMonkey is evaluated on Google Octane [2], which is a benchmark suite used to measure a JavaScript engine's performance by running a suite of test applications. Our experiments cover 12 of 15 applications in Octane, due to the failure of 3 applications when run on original HQEMU.

The experiment platform is equipped with Intel(R) Xeon CPU 16 cores with Hyper-threading enabled, 64G bytes memory, and Ubuntu-14.04 with Linux-3.13. To reduce the influence of random factors, each application is run three times and their arithmetic average is taken.

4.1 Performance Improvement

Figure 6 shows the performance improvement achieved by using our persistent code caching approach. The baseline is the execution without helper thread and persistent code, i.e., `w/o H + w/o P`. In this experiment, different numbers of helper threads were evaluated, but only the performance results with one helper thread are presented here because their results are all very similar. On

	P(capture)	P(connect)	P(connect_rot)	P(connection)	P(connection_rot)	P(cutstone)	P(dniwog)
capture	100%(4)	90.73%(2.5)	83.04%(2.4)	91.16%(2.3)	76.03%(2.6)	88.22%(2.3)	89.23%(2.2)
connect	66.36%(2.4)	100%(3.5)	76.03%(2.3)	88.58%(2)	71.06%(2.3)	67.25%(2.2)	86.33%(2)
connect_rot	73.7%(2.4)	91.9%(2.3)	100%(4)	91.91%(2.2)	83.15%(2.4)	74.38%(2.3)	90.86%(2.2)
connection	43.48%(2.2)	57.48%(2)	49.51%(2.2)	100%(3.5)	47.82%(2.3)	50.81%(2)	84.02%(2.1)
connection_rot	70.18%(2.6)	89.27%(2.3)	86.56%(2.4)	92.17%(4.2)	100%(4.2)	70.61%(2.6)	91.12%(2.4)
cutstone	73.97%(2.3)	76.89%(2.2)	70.31%(2.3)	89.40%(2)	64.25%(2.6)	100%(3.6)	88.65%(2)
dniwog	28.83%(2)	37.53%(2)	32.94%(2.1)	56.46%(2)	31.83%(2.2)	33.98%(1.9)	100%(3.3)

Table 1: Persistent code hit ratios with different inputs (the numbers in parentheses are the average numbers of basic blocks in the hit traces of persistent code).

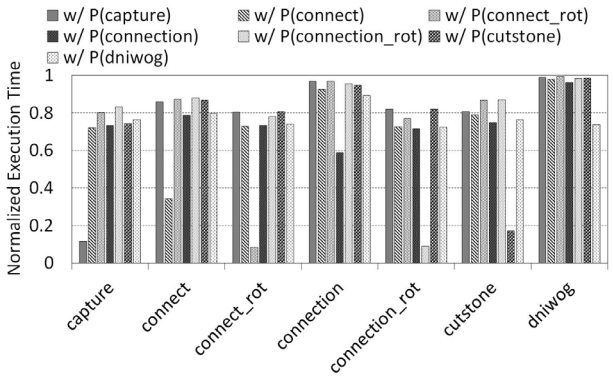


Figure 7: Effectiveness of persistent code for the same application with different inputs.

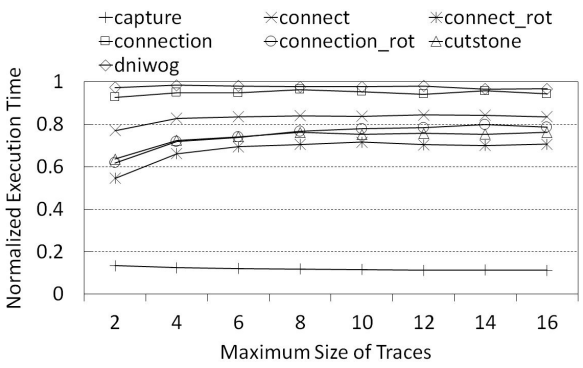


Figure 8: Performance sensitivity of persistent code caching on the maximum size of traces.

average, 76.4% and 9% performance improvements are achieved compared to the original DBT system without and with a helper thread enabled, respectively.

As shown in Figure 6, the performance is improved for all applications except `400.perlbench` with `test.pl` input. For this application, our approach achieves 40% performance improvement without helper threads, but introduces 5.5% overhead with helper threads. After further investigation, we have found that this input creates many child processes (around 72 of them) to perform many extremely short-running tests.

In the execution with both persistent code and helper threads, our implementation looks up persistent code before sending a trace translation request to helper threads. The performance overhead introduced by this look-up process might be relatively high for these extremely short-running tests because the generated persist code is very large (more details are presented in Section 4.3). However, in the execution with helper threads but without persistent code, a trace translation request is sent to helper threads once after the trace is formed, which introduce very little performance overhead. This is also the reason why it cannot achieve similar performance improvement in this application for `w/o H + w/ P` compared to `w/ H + w/o P`. Similar characteristics can be observed on several applications in Octane, e.g., `Crypto`, `PdfJS`, and `CodeLoad`, but they have different causes. More details are discussed in Section 4.2.

To study the efficiency of persistent code across different inputs for the same application, an experiment is performed on `445.gobmk`, which plays the game of Go, a game with simple rules but has extremely complicated strategies. There are seven inputs for this benchmark contained in the *test* input.

In this experiment, persistent code generated from one input is reused in a run using another input. The experimental result is illustrated in Figure 7, where `w/ P(X)` means persistent code generated from `X` input is used. Helper thread is disabled in this experiment. As shown in the figure, persistent code is still helpful to improve the performance across different inputs. This is because different inputs of the same application are very likely to share same blocks/traces. Table 1 shows the percentage of blocks/traces that can be found in persistent code, i.e., persistent code hit ratio. The numbers in the parenthesis are the average sizes of the hit traces in persistent code. This size is more than 3 for the same input. However, it is less than 3 for the different inputs. This means, the average size of traces shared across different inputs of the same application is usually not too large.

The default maximum size of a trace is 16 basic blocks in the original HQEMU. Another experiment is con-

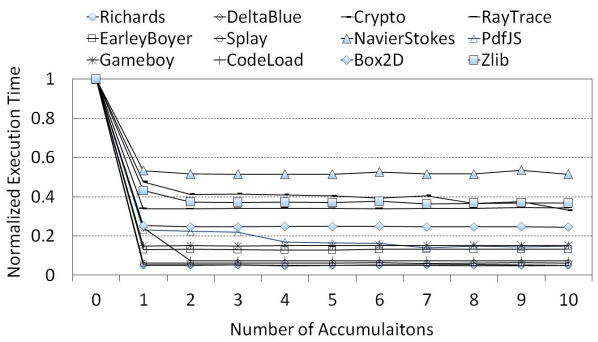


Figure 9: Effectiveness of persistent code accumulation from multiple runs of the *same* application.

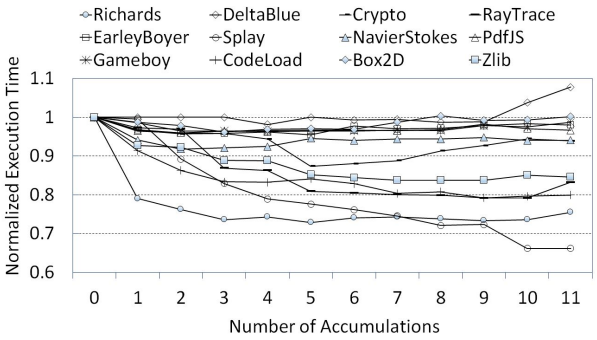


Figure 10: Effectiveness of persistent code accumulation from multiple runs of *different* applications.

ducted to study the performance sensitivity of persistent code on the size of traces across different inputs. In this experiment, the input `capture` is used for generating persistent code, and the maximum size of a trace is set between 2 and 16. The persistent code generated in each maximum trace size is applied to other inputs with the same maximum trace size. Due to the potential impact of the maximum size on the performance, various baselines of each input are used depending on the chosen maximum size of traces. As shown in Figure 8, the best performance is achieved when the maximum size of traces is 2. This shows why the average size of hit traces in the last experiment is less than 3. Hence, code persistence across different inputs prefers to use shorter traces (2 or 3 basic blocks).

4.2 Accumulation Effectiveness

The effectiveness of persistent code accumulation is evaluated on the same and different applications. Due to their different behavior, C/C++ and JavaScript applications in our benchmark suite were evaluated separately. Here, we only show the experimental results of JavaScript applications because C/C++ applications have similar results and our space is limited here.

Firstly, we evaluate the effectiveness of persistent code accumulation for the *same* application. In this experiment, the persistent code is accumulated from multiple runs for the same application. Each application is executed multiple times with previously generated persistent code accumulated and used in the later runs, if possible. Each time when a new persistent code is encountered, it is included for the later runs of the same application to evaluate the benefit of the accumulation.

Figure 9 shows the performance (normalized to the execution time without using persistent code) of each application using persistent code accumulated from one run up to ten runs. As shown in the figure, such persistent code accumulation is helpful to several applications, e.g., `Crypto`, `PdfJS`, and `CodeLoad`. However, the benefit of accumulation quickly diminishes after a small number of runs for most applications. The figure also shows that persistent code cannot achieve similar performance improvement for `w/o H + w/ P` compared to `w/ H + w/o P` because persistent code collected from one execution cannot fully cover the later executions.

Typically, there are two reasons for this phenomenon. (1) There is a chance to miss some traces due to the limited buffer size used for trace formation in HQEMU. These traces have opportunities to be formed and translated in the executions when persistent code is available. (2) The behavior of guest JIT engine is changed due to the performance improvement introduced by persistent code. In this situation, new guest binary is generated dynamically by the guest JIT engine that requires translation. Generally, the performance becomes stable after two or three accumulations for most applications shown in Figure 9 because no significant amount of new host code is generated beyond that point.

Next, we evaluate the effectiveness of persistent code accumulation from *different* applications. In this experiment, the persistent code is accumulated from the executions of all applications in a set of twelve applications except for one. The accumulated persistent code is then used by the excluded application. To evaluate the effectiveness of accumulation, the excluded application is run with the persistent code accumulated from one application up to eleven applications in the set. Without loss of generality, the order of the eleven applications chosen during the accumulation phase is random.

Figure 10 shows the results of this experiment. Initially, most applications can benefit from the accumulated code. However, as more persistent code is accumulated (from one to eleven), its performance benefit diminishes and even becomes harmful beyond some point for some applications. This indicates that persistent code accumulation from different applications is not always beneficial because large persistent code can introduce high

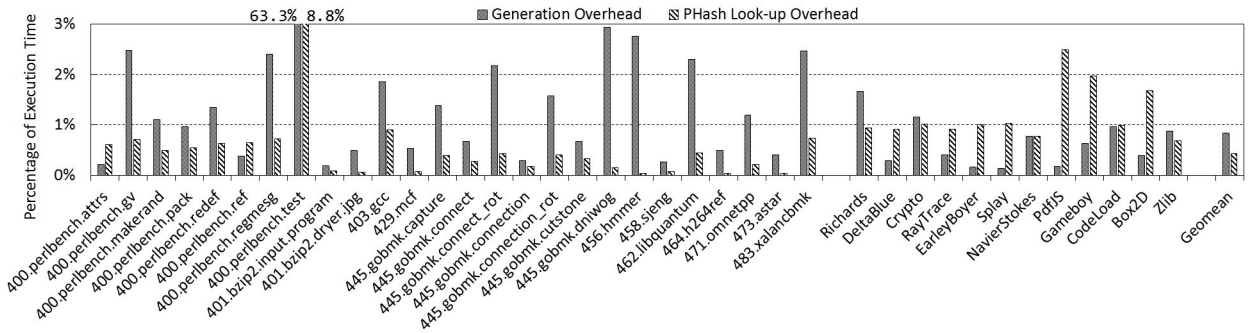


Figure 11: Performance overhead introduced by persistent code caching.

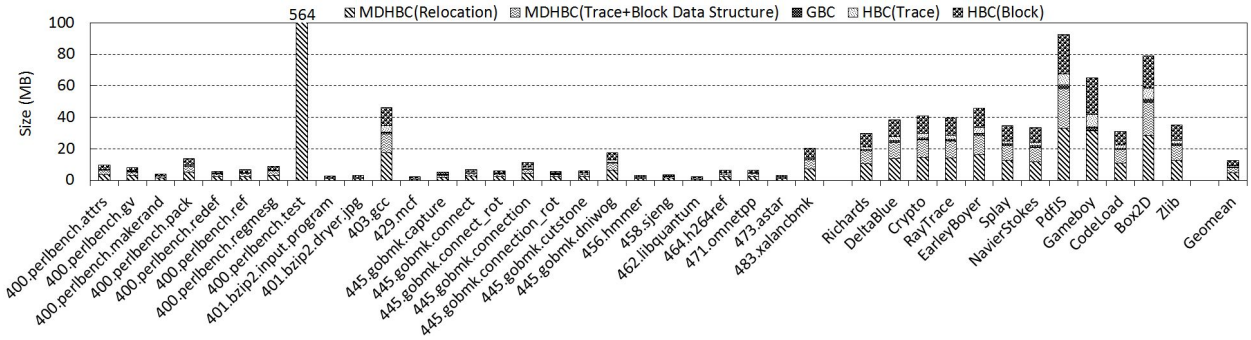


Figure 12: The sizes of different parts in persistent code.

overhead in looking up PHash. In our experiments, 5 is the turning point. Currently, there is no limitation on the size of accumulated persistent code. An effective persistent code management approach should limit the size to avoid this detrimental impact.

4.3 Performance Overhead and Code Size

Figure 11 shows the performance overhead introduced by persistent code generation and PHash lookup. The baseline is the original execution without helper thread and without persistent code. As shown in the figure, for all applications except `400.perlbench` with `test.pl` input, both kinds of overhead are less than 3%. As mentioned before, `400.perlbench` with `test.pl` has multiple child processes. Due to the persistent code accumulation from each child process, a huge generation overhead is introduced. After analyzing the execution, we found 95% of the performance overhead is caused by disk file operations. Another overhead is from PHash lookup because of the large accumulated persistent code. On average, less than 1% performance overhead is introduced by our persistent code caching approach.

Figure 12 shows the size of each part of the persistent code. For most applications, the total size is less than 40MB, except `400.perlbench` with `test.pl` input, which generates a very large persistent code due to persistent code accumulation. As shown in the figure, MDHBC is the largest part in persistent code, with 61% on average. In our current implementation, all information in the internal data structures of blocks/traces is kept for future recovery. However, some of information in these data structures is never used. There still is a good potential to further reduce the size of persistent code, and is part of our future work. The second largest part in persistent code is HBC(Block), which occupies 28% of the memory space, compared to 6.8% for HBC (Trace). This is because the host binary of traces is translated and optimized by LLVM JIT, while host binary for basic blocks is translated by TCG and is not optimized.

5 Related Work

The potential of reusing translated code across executions have been studied in previous work [12, 21, 22, 7]. Using DynamoRIO [4], which is a dynamic binary optimization system, the work [12] shows that many of the most heavily executed code traces in SPEC CPU2000 are similarly optimized during successive executions. This indicates the significant potential for leveraging the inter-execution persistence of translated/optimized code.

A mechanism of persistent code cache has been im-

plemented in [21, 22] for Pin [17], which is a dynamic binary instrumentation system. It takes traces and corresponding internal C++ data structures used by Pin as persistent code, which are generated when the original code cache in Pin becomes full or at the exit of the execution. To reuse persistent code across applications, shared libraries have to be loaded to the same memory address. Another persistent code caching scheme is proposed for process shared code cache [7]. It organizes guest applications and libraries into separate modules, and shares the code cache of modules among processes. To support relocatable guest applications, an offset of an instruction address from the start of the module is used to look up the persistent code of this module.

Compared to those schemes, our approach has three significant advantages. First, the guest binary code rather than the guest instruction address or offset is used to look up persistent code. It enables our approach to support dynamically generated code. Second, there is no restriction on the change of guest application and the libraries it depends on. If the guest application (or the libraries) is modified since last time when the persistent code was generated, e.g., upgraded to a newer version, our persistent code can still benefit from the unchanged part in guest binary code without any modification. Lastly, persistent code generated from a static-linked application, which contains application code and library code, is still helpful for other applications (static-linked or dynamic-linked) that use the same library code. However, it is difficult to realize this in existing mechanisms.

A similar scheme of guest binary matching, called guest binary verification, is also explored to reuse translated code in a single execution [16]. Different from the approach discussed in this paper, there is no relocation issue in a single execution. Also, it uses guest instruction address to discover persistent code, which is different from our approach using guest binary code.

There are also several schemes to optimize DBT systems [11, 26, 27, 10, 23]. Basically, these optimizations can cooperate with our persistent code caching approach to improve the performance of DBT systems.

6 Conclusion

This paper presents a general and practical persistent code caching framework to work within existing DBT systems. The proposed approach saves translated host code as persistent code and reuses it across executions to amortize or mitigate the translation overhead. Different from existing persistent code caching schemes, our proposed approach uses guest binary code to look up and verify persistent code. One significant benefit from this approach is to support persistent code caching for dynamically generated code, which is very popular in script languages. To find out a matching persistent code entry, a two-level hash table is designed to organize persistent code entries and speed up the look-up process. A prototype of this approach has been implemented in an existing retargetable DBT system. Experimental results on a set of benchmarks, including C/C++ and JavaScript, show that this approach can achieve 76.4% performance improvement on average compared to the original DBT system without helper threads to offload the overhead of dynamic binary translation, and 9% performance improvement on average over the same DBT system with helper threads when translated code reuse is combined with the help threads.

7 Acknowledgments

We are very grateful to Andy Tucker and the anonymous reviewers for their valuable feedback and comments. This work is supported in part by the National Science Foundation under the grant number CNS-1514444.

References

[1] ART and Dalvik. `https://source.android.com/devices/tech/dalvik/index.html`.

[2] Google Octane. `https://developers.google.com/octane`.

[3] Mozilla SpiderMonkey. `https://developer.mozilla.org/en-%US/docs/Mozilla/Projects/SpiderMonkey`.

[4] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2000), PLDI '00, ACM, pp. 1–12.

[5] BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium®-based Systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2003), MICRO 36, IEEE Computer Society, pp. 191–201.

[6] BELLARD, F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATC '05, USENIX Association, pp. 41–46.

[7] BRUENING, D., AND KIRIANSKY, V. Process-shared and Persistent Code Caches. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 61–70.

[8] CHEN, J.-Y., SHEN, B.-Y., OU, Q.-H., YANG, W., AND HSU, W.-C. Effective Code Discovery for

ARM/Thumb Mixed ISA Binaries in a Static Binary Translator. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (Piscataway, NJ, USA, 2013), CASES '13, IEEE Press, pp. 19:1–19:10.

[9] CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. FX!32: A Profile-Directed Binary Translator. *IEEE Micro 18*, 2 (Mar. 1998), 56–64.

[10] FEINER, P., BROWN, A. D., AND GOEL, A. Comprehensive Kernel Instrumentation via Dynamic Binary Translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 135–146.

[11] HAWKINS, B., DEMSKY, B., BRUENING, D., AND ZHAO, Q. Optimizing Binary Translation of Dynamically Generated Code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2015), CGO '15, IEEE Computer Society, pp. 68–78.

[12] HAZELWOOD, K., AND SMITH, M. D. Characterizing Inter-Execution and Inter-Application Optimization Persistence. In *Workshop on Exploring the Trace Space for Dynamic Optimization Techniques* (2003), pp. 51–58.

[13] HONG, D.-Y., HSU, C.-C., YEW, P.-C., WU, J.-J., HSU, W.-C., LIU, P., WANG, C.-M., AND CHUNG, Y.-C. HQEMU: A Multi-threaded and Retargetable Dynamic Binary Translator on Multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (New York, NY, USA, 2012), CGO '12, ACM, pp. 104–113. *See* `http://itanium.iis.sinica.edu.tw/hqemu/`.

[14] LATTNER, C. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.org`.

[15] LEE, G., PARK, H., HEO, S., CHANG, K.-A., LEE, H., AND KIM, H. Architecture-aware Automatic Computation Offload for Native Applications. In *Proceedings of the 48th International Symposium on Microarchitecture* (New York, NY, USA, 2015), MICRO-48, ACM, pp. 521–532.

[16] LI, J., ZHANG, P., AND ETZION, O. Module-aware Translation for Real-life Desktop Applications. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (New York, NY, USA, 2005), VEE '05, ACM, pp. 89–99.

[17] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), PLDI '05, ACM, pp. 190–200.

[18] NETHERCOTE, N., AND SEWARD, J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2007), PLDI '07, ACM, pp. 89–100.

[19] OTTONI, G., HARTIN, T., WEAVER, C., BRANDT, J., KUTTANNA, B., AND WANG, H. Harmonia: A transparent, efficient, and harmonious dynamic binary translator targeting the intel® architecture. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (New York, NY, USA, 2011), CF '11, ACM, pp. 26:1–26:10.

[20] RATANAWORABHAN, P., LIVSHITS, B., AND ZORN, B. G. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development* (Berkeley, CA, USA, 2010), WebApps'10, USENIX Association, pp. 3–14.

[21] REDDI, V. J., CONNORS, D., COHN, R., AND SMITH, M. D. Persistent Code Caching: Exploiting Code Reuse Across Executions and Applications. In *Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), CGO '07, IEEE Computer Society, pp. 74–88.

[22] REDDI, V. J., CONNORS, D., AND COHN, R. S. Persistence in Dynamic Code Transformation Systems. *SIGARCH Comput. Archit. News 33*, 5 (Dec. 2005), 69–74.

[23] ROY, A., HAND, S., AND HARRIS, T. Hybrid binary rewriting for memory access instrumentation. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2011), VEE '11, ACM, pp. 227–238.

[24] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[25] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. HDTrans: An Open Source, Low-level Dynamic Instrumentation System. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), VEE '06, ACM, pp. 175–185.

[26] WANG, C., WU, Y., BORIN, E., HU, S., LIU, W., SAGER, D., NGAI, T.-F., AND FANG, J. Dynamic Parallelization of Single-threaded Binary Programs Using Speculative Slicing. In *Proceedings of the 23rd International Conference on Supercomputing* (New York, NY, USA, 2009), ICS '09, ACM, pp. 158–168.

[27] WANG, W., WU, C., BAI, T., WANG, Z., YUAN, X., AND CUI, H. A Pattern Translation Method for Flags in Binary Translation. *Journal of Computer Research and Development 51*, 10 (2014), 2336–2347.

Instant OS Updates via Userspace Checkpoint-and-Restart

Sanidhya Kashyap Changwoo Min Byoungyoung Lee Taesoo Kim Pavel Emelyanov†
Georgia Institute of Technology †*CRIU & Odin, Inc.*

Abstract

In recent years, operating systems have become increasingly complex and thus more prone to security and performance issues. Accordingly, system updates to address these issues have become more frequently available and increasingly important. To complete such updates, users must reboot their systems, resulting in unavoidable downtime and further loss of the states of running applications.

We present KUP, a practical OS update mechanism that employs a userspace checkpoint-and-restart mechanism, which uses an optimized data structure for checkpointing on disk as well as a memory persistence mechanism across the update, coupled with a fast in-place kernel switch. This allows for instant kernel updates spanning across major kernel versions without any kernel modifications.

Our evaluation shows that KUP can support any type of real kernel patches (e.g., security, minor or even major releases) with large-scale applications that include `memcached`, `mysql`, or in the middle of the Linux kernel compilation, unlike well-known dynamic hot-patching techniques (e.g., `ksplice`). Not only that, KUP can update a running Linux kernel in 3 seconds (overall downtime) without losing 32 GB of `memcached` data from kernel version v3.17-rc7 to v4.1.

1 Introduction

Today, computer users routinely update their operating systems either to patch security vulnerabilities, fix bugs, or add a new set of features to the existing system. Unfortunately, to fully incorporate a new update, users have no choice but to restart their systems, which results in the loss of the running states of applications. This unavoidable disruption not only brings inconvenience to end users, but also causes an adverse financial impact on business.For example, updating a `memcached` [57] server at Facebook that caches around 120 GB data in 144 GB RAM [24] requires a prolonged warm-up phase of 90-100 minutes [30]. The more critical problem is when a system update fails. Such failure often leads to additional downtime or maintenance costs thereafter, resulting in an immediate loss of active customers [20, 21]. Still, even with the unavoidable disruption and risks involved in update failures, system updates are necessary to promptly mitigate the known security issues and resolve critical bugs that might hurt the correctness of operations [8, 25, 44, 68, 71].

To solve these problems, two large sets of techniques are used in practice: dynamic hot-patching and rolling updates. Dynamic hot-patching (or live update) [4, 67, 73] directly applies patches to the running kernel. As a result, system updates can be performed without incurring application downtime. However, dynamic hot-patching is inherently limited to patches that do not semantically modify data structures. Therefore, it is commonly used to apply simple security patches that contain minor code changes. For instance, `kpatch` [67], a popular dynamic update tool developed by Red Hat, was able to support 2 out of 23 minor updates over a year of Ubuntu's kernel releases (see Figure 1).

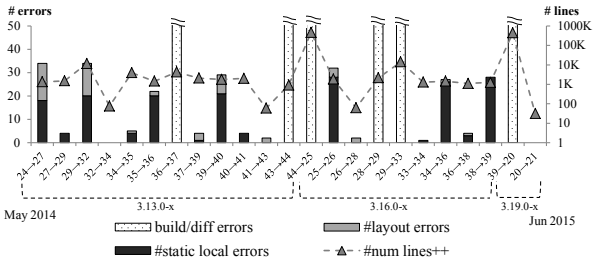


Figure 1: Limitation of dynamic kernel hot-patching using `kpatch`. Only two successful updates (3.13.0.32→34 and 3.19.0.20→21) out of 23 Ubuntu kernel package releases. X-axis represents before-version and after-version, and dotted bars represent failures in executing `kpatch`. Each failure case represents the following legends: build/diff errors—`kpatch` failed during a build and diff processes; # static local—a patch modifies a value of static local variables; # layout error—a patch changes a layout of a data structure.

Rolling updates [20, 60] are another viable technique for large-scale systems. In rolling updates, system administrators apply an update to a small group of machines; if there is no failure, they apply the same update to the rest of the machines in the data center. This helps administrators minimize the risk of update failures and system downtime. However, this process requires careful planning to minimize the disruption of running services due to the inevitable downtime of applications.

There are several research projects that attempt to solve the OS update problem either by proposing new OS construction [5, 7, 23, 65, 80], or by providing a transfer function for system updates [28]. In order to keep their internal states switchable, such OSes require radical design changes, thereby making these modifications impractical for the commodity OSes [19, 69].

To address these issues, we present KUP—a new update mechanism that allows for prompt kernel updates without modifying a commodity OS. KUP incorporates application checkpoint-and-restart (C/R) for saving and restoring the application states before and after the update,

and an in-place kernel switch to quickly boot into a new kernel. This allows KUP to minimize the downtime of running applications during the system update. Unlike dynamic hot-patching mechanisms, KUP can perform a full system update regardless of the complexity of updates, and thus can support a broader range of system updates, including security and bug fixes, the addition of new features, and even major kernel upgrades. To address update failures, KUP provides a safe fallback mechanism that provides a mechanism to restore the original system, checkpointed right before the update.

KUP works by leveraging the OS provided infrastructure such as `procfs`, `ptrace` system call [37] for obtaining information during the checkpoint of an application, and process forking (`clone`, `fork`) mechanism along with other system calls [15, 37] and `netlink` sockets to restore the application after update. To enable an efficient, userspace-based C/R of an application during an update, we design an efficient checkpointing storage format that improves both the checkpoint size and KUP's performance. However, the aforementioned solution of KUP still ends up storing RAM data on the disk, which is not suitable for disk-less systems [30, 57].[1] We further extend KUP and make it generic enough to support disk-less systems by proposing a memory persistence mechanism, which we implement via binary patching and dynamic OS instrumentation. This approach is complementary to our userspace solution.

This paper makes the following three contributions:

- We design a simple, yet robust update mechanism by using application C/R, and implement an open source prototype of KUP.
- We show KUP's effectiveness by providing an in-depth analysis of the proposed techniques with real applications, micro-benchmarks and full software stack representing generic data center application.
- We devise, implement and evaluate a safe fallback mechanism for KUP that can be easily realized through application C/R and an in-place kernel switch.

The rest of this paper is organized as follows. §2 compares KUP with previous work. §3 gives the high-level ideas of KUP's approach to instant kernel update. §4 points out the challenges and §5 describes our design. §6 explains our implementation and §7 evaluates KUP's performance. Lastly, §8 discusses its limitations and potential optimizations and §9 concludes.

[1] The `memcached` servers at Facebook are disk-less and they keep everything in memory.

2 Related work

In this section, we compare KUP's approach to previous studies in four areas: dynamic hot-patching, new OS designs, live migration, and application C/R for updates.

Dynamic hot-patching. Industries rely on solutions such as `Ksplice` [4], `kpatch` [67], and `KGraft` [73] to promptly mitigate the security vulnerabilities without any significant outage of their production systems. However, as shown in Figure 1, the hot-patching techniques have critical limitations, especially in handling data layout changes, thus making it impossible to guarantee the safety of live updates. Previous studies have proposed dynamic software update schemes on event-driven systems [1, 27, 28, 36], object oriented languages [39, 63] and even C language [4, 35, 53, 54, 56] including formal proofs [34, 55, 72].

However, unlike previous update techniques, KUP relies on a whole kernel switch to apply any kind of patch, regardless of its complexity. KUP allows for system-wide live updates without modifying programs or tracking their state changes, unlike previous works [19, 35].

New operating system designs for live update. Another approach is to design a new OS with well-defined abstractions between interfaces and implementations so that each component can be replaced online without disruption. Representative examples include Microkernel (e.g., exokernel [23], K42 [5, 7, 70], Barrelfish [6, 80]) and LibOS (e.g., Drawbridge [65]). In particular, Proteos [28] based on MINIX 3, and a Linux variant [69], are designed to update their components online and also transform internal data structures to adopt the updated kernel. Unfortunately, with a large amount of code changes (e.g., new features), constructing the state transfer functions either requires manual effort or is infeasible in many cases [5, 7, 11, 28, 69, 70].

Autopod [66] and the work by Siniavine et al. [69] are closest to KUP. Autopod uses a namespace mechanism for process decoupling and migrating applications across machines, whereas Siniavine et al. provides a kernel space C/R mechanism coupled with an in-place kernel switch to update the OS, and employs techniques similar to those of Otherworld [19]. The work by Siniavine et al. heavily modifies various kernel subcomponents to checkpoint and restore the applications entirely in the kernel space without any user intervention. In contrast, Otherworld tries to recover applications from the kernel failures by transferring them to the new kernel via an in-place kernel switch.

Unlike these studies, KUP achieves its goal of instantly updating the systems via userspace C/R and an optimized in-place kernel switch *without modifying the kernel source*. KUP faces a different set of challenges, such as (a) how to efficiently checkpoint storage data structure for

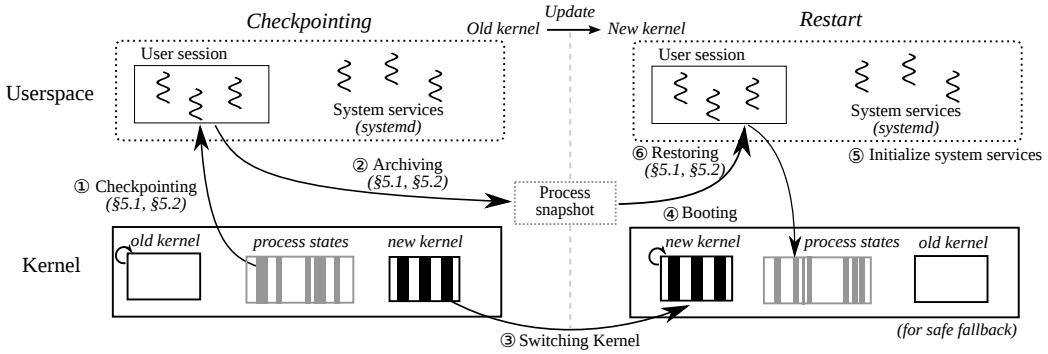


Figure 2: Overview of KUP's updating procedures. KUP first checkpoints user's processes ①, and archives their snapshots ②. After checkpointing selected processes in a user's current session, KUP replaces the old kernel to the new kernel image ③, and finally switches to the new kernel ④. After the new kernel boots, KUP first initializes its system daemons ⑤, and finally restores snapshots of user applications ⑥.

effectively using both checkpoint and restore techniques, and (b) how to implicitly modify the kernel functionality to remove the redundant memory copy phase from the application C/R, which we do via a binary patching mechanism. KUP leverages the kernel-exposed userspace information to extract the kernel-resident data-structures' data via `procfs`, netlink sockets and system calls [15]. This makes the process checkpointing independent of both the kernel update and the checkpoint format.

Application C/R. The area of application C/R has been explored in various contexts: object-oriented languages [50, 77], single process [61, 64], multiple processes [2, 49], even across multiple computers in a distributed environment [43, 79]. It has also been explored solely for an optimized restoration scheme [81]. In practice, application C/R has been used for bootstrapping system startup [51], achieving fault-tolerance and performance in HPC [26, 33], debugging [61, 77], and even for recovering system integrity [45–47].

Unlike these approaches, KUP can retrofit the known application C/R scheme for live updates. Currently, it relies on `criu` [22] for application C/R because of its mature code base and stable checkpointing mechanism without modifying the kernel. General application C/R approaches focus on reducing the quiescence period, but KUP suffers from a memory snapshotting bottleneck, since the kernel provides the quiescence mechanism via the `ptrace` system call [37] for KUP to leverage. KUP uses incremental checkpoint [64] and on-demand restore [58] to reduce the downtime during application C/R. Unlike libckpt [64], KUP does not require any modifications to applications and uses a new file format to effectively merge both techniques.

Live update with virtual machine migration. The internal data structures of operating systems can be captured by providing underlying abstractions such as virtual machine [12, 38] or namespace containers [41, 48]. Microvisor [52] allows for both live update and rolling update of host and guest operating systems by migrating a suspended virtual machine to another host before proceeding with an update. On the other hand, ShadowReboot [78] updates the guest OS by forking and restoring the snapshot of the old guest state on the same host.

Unlike these approaches, KUP directly enables the in-place live update of an OS without relying on a separate host. This allows KUP to be easily used on clusters with heterogeneous hardware, where live migration is problematic due to the incompatibility of CPUs and devices.

3 KUP Life Cycle

KUP's goal is to update an OS (e.g., kernel and system services) by immediately applying patches without incurring observable downtime. KUP achieves this goal by first preserving the states of running applications and then restoring their previous states on the updated OS. This will allow administrators to update systems with minimal downtime. Importantly, this update procedure can benefit end users, by allowing for frequent updates not only for immediate security fixes, but also for less critical changes, such as performance improvements and the incorporation of new functionality (e.g., new I/O scheduling policy and also software rejuvenation at the OS level [40]).

Conceptually, KUP's update procedure consists of three steps. It first checkpoints the running applications, then switches to the updated OS, and finally restores the suspended applications. This section describes KUP's update procedure followed by a scenario enabled by KUP (i.e., safe fallback). Figure 2 gives a holistic overview of KUP's update cycle for applications.

Checkpointing applications. Before updating the OS, KUP takes a snapshot of running applications. This snapshot is made up of process states, consisting of their memory space (e.g., code/data sections and stack/heap, etc.) and their internal states in the kernel, including for example the states of sockets, etc. (① checkpointing in Figure 2). The snapshot is then stored in a persistent stor-

age, where KUP can fetch it after switching to the updated OS (② archiving).

Switching kernel. After checkpointing, KUP first loads the new kernel binary to its memory. Then, it switches its running kernel (old version) to the new version that was just loaded (③ switching kernel). The new kernel quickly boots up by skipping most of the hardware initialization routine (such as slow BIOS and POST) as the devices and peripherals were already initialized ④. The system management daemon relaunches the required system services after the kernel initialization ⑤. KUP never checkpoints the system services, since they are already maintained by the system management daemon.

Restoring applications. After running the newly updated kernel along with its system services, KUP starts the process of restoring the user's applications that were suspended before the update. It reconstructs the process-specific states inside the kernel and then restarts the application (⑥ restoring).

The application C/R and kernel switch not only allow for a seamless kernel update, but also facilitate a mechanism to safely fall back to the previous kernel in case of an update failure.

Safe fallback. KUP also supports safe fallback, a mechanism that allows for automatic fallback to the previous OS state. With this mechanism, KUP tries to preserves the availability of the older kernel to tackle failure cases introduced by a buggy updated kernel as well as the liveness of the applications.

4 Challenges

The existing general C/R tools can efficiently checkpoint and resurrect applications provided by the kernel-exposed userspace information due to the inclusion of a namespace-based container mechanism [14, 32, 41]. However, they suffer from two well known issues: *quiescence* and *memory checkpointing*. Quiescence is a time span in which all the kernel-resident data structures and the states of threads at the kernel and user levels are consistent with the checkpointed image. These both lead to application downtime during C/R. In our experiments, the quiescence time period constitutes only 0.01%–2% of the total downtime while updating the OS with KUP, whereas the major downtime occurs because of the memory dump (95%–99%). Hence, we focus on understanding the impact of the memory dump during the update as well as the inherent limitation of the userspace C/R from an OS update perspective. Later, we will briefly discuss how KUP achieves quiescent state in §6 and its contribution to the downtime.

Memory serialization for application C/R. We will first give a brief background of the existing state-of-the-art C/R techniques which KUP use:

- **Incremental checkpointing.** Checkpointing the memory of processes imposes significant downtime when using existing userspace application C/R tools. The downtime will be unacceptable for applications like `memcached`, since for these it is preferable for the application's data to be persistent and usable after an update. To resolve this issue, the C/R component of KUP implements *incremental* checkpointing. This idea itself is not entirely new [12, 64], but it reshapes this as an application-agnostic userspace C/R mechanism: it takes multiple *asynchronous* snapshots of the process's memory, followed by a *synchronous* one. This leads to minimal downtime. KUP only suspends the process in the last iteration, because of the minimal checkpointed data in the last *asynchronous* iteration.
- **On-demand restore.** The naive restore mechanism of the existing C/R tools imposes equivalent downtime, similar to the naive checkpoint mechanism. For example, sequentially reading the checkpointed data from the disk imposes an unacceptable downtime for the memory intensive applications, like `memcached`, during the restore phase. To resolve this issue, the C/R component of KUP starts the process without loading its entire memory contents, instead it reloads them *on-demand* when the process tries to access a particular memory address. This accelerates the process restart, hence decreasing the application's downtime during restore.

However, the coupling of both on-demand restore and incremental checkpoint degrades the overall performance after an update. In particular, this coupling suffers from two problems: (1) the overhead of finding a corresponding file page when a page fault occurs in a checkpointed snapshot image, and (2) the overhead incurred by `mmap()` when binding each reloaded page in userspace. Note that performance and scalability problems of `mmap` operations in the Linux kernel are well known [13].

Eliminating redundant memory copy. Even though the merging of the aforementioned C/R techniques speeds up the C/R process, it copies the memory twice between the kernel and the user space. Furthermore, the backend storage, which keeps the snapshot persistent across system updates, becomes the bottleneck for disk-less systems. For example, it is not possible to update Facebook's diskless `memcached` servers, since they cache 120 GB of data in memory and do not have any storage medium available for the warm-up phase.

5 KUP Design

KUP's design goals are to (1) instantly update a running kernel, (2) avoid any observable downtime, (3) ensure no kernel modification, and (4) supporting all types of

patches, unlike previous approaches [19, 69]. To realize these goals while addressing the aforementioned challenges, we first present a new data structure called FOAM to efficiently serialize memory for both C/R techniques (§5.1). Then we present another optimization technique, called *persistent physical pages* (PPP), to further decrease the downtime during kernel update (§5.2). Later, we introduce KUP's safe fallback mechanism.

5.1 Userspace Application C/R

By using incremental checkpoint, the C/R component of KUP suffers from slower restoration than the naive checkpointing. This results in slower application restart due to its extra work in maintaining a sequence of images taken during the incremental checkpoint. Moreover, the performance further degrades when the incremental checkpointing is combined with on-demand restore. For example, depending on the application's writable working set, memory address ranges (e.g., dirty pages) as well as their respective metadata (e.g., previous page and address region) get fragmented and scattered across multiple checkpointed dumps (files). Hence, the C/R component needs to install a huge number of mappings to backup *each page* by the checkpointed dump in the restore phase. This imposes non-negligible overhead for enabling the on-demand restore with the incremental checkpoint, and also results in lots of context switches while installing the mappings at the page granularity.

To solve this problem, we design a simple yet effective data structure called *file offset-based address mapping* (FOAM), for quickly restoring the processes' pages from the checkpointed images. FOAM uses a direct one-to-one mapping between the process address and file offset in a *sparse* file. Due to this, KUP avoids the maintenance cost of file indexes and fine-grained pages. Instead, KUP can bind the checkpointed image to the whole address space.

We prepare this sparse file large enough to represent 64-bit virtual address spaces ($2^{48-1} = 128$ TB excluding kernel and canonical address spaces). We leverage the concept of *holes*, which all modern filesystems support, such that a single file is large enough to express whole virtual address spaces (e.g., 16 EB support by `xfs` and `btrfs`)[2] while never occupying such a huge amount of physical storage spaces underneath.

FOAM's approach brings three advantages to KUP for userspace-based OS update: (1) it eliminates the overhead to maintain metadata by directly mapping process address space to the file offset (e.g., mapping virtual memory region at page granularity); (2) it resolves the data fragmentation issue from the incremental checkpointing (e.g., new snapshot might contain a newly updated memory region if its size is changed from its previous snapshot); (3) it simplifies the design to enable the on-demand restore (e.g., mapping a single file to the entire process space). With FOAM, KUP can leverage the incremental checkpoint and on-demand restore together, thereby reducing the downtime caused by application C/R.

[2] FOAM is not limited by a file system that does not support such a huge filesize. For example, to support `ext4`, KUP can serialize virtual address spaces into 8 files for each process as `ext4` supports 16 TB file size.

5.2 Reusing Persisted Memory Across Update

Even though the FOAM approach is able to decrease the downtime experienced by application by using state-of-the-art C/R techniques, it still suffers from redundant copying of the process's memory before and after the update. Also, there should be sufficient space to save the checkpointed data. To thwart these problems, KUP introduces a new mechanism called *persistent physical pages* (PPP). PPP removes the redundant copying of the memory from the kernel to the userspace and vice-versa. Instead, it preserves the process' memory across the system update.

PPP first saves the virtual address of a process and its corresponding physical mapping at the page granularity. Then, while booting the new kernel, KUP reserves the mapping information and the corresponding pages to forbid their usage. Later, KUP relies on the page fault handler to rebind the pages with the restoring application in an on-demand fashion. This allows KUP to reuse the same physical pages at the time of restoration after the update. This technique is effective in two ways: (1) it avoids redundant copies of process' memory during application C/R, rather preserving them without extra overhead; (2) it allows for an instant, page-level on-demand restoration of a process. Unlike any application C/R, PPP does not degrade the performance of the restored process, as it quickly rebinds the physical page upon a page fault.

Currently, the kernel does not provide any functionality to completely implement PPP in the userspace. The previous work [69] has heavily modified multiple kernel subcomponents to achieve it entirely in the kernel space. On the contrary, keeping KUP's design goals in mind, we instead hook the kernel's memory management unit with the binary patching mechanisms, such that PPP is easily applicable to the current commodity OSes.

To implement PPP, KUP needs to reserve the set of pages as well as the virtual-to-physical mapping information after the kernel update and then handle the page faults for the restoring process. Since both steps are only possible in the kernel space, we use static binary instrumentation for injecting the code in the kernel binary image and dynamic kernel instrumentation to hook the page fault handling functionality. Next, we describe binary patching in detail, and later the steps to perform PPP.

Binary patching. For memory reservation of pages and their mappings, KUP performs binary patching on the new kernel binary in which the system will boot into. Before starting the checkpointing process, KUP first does the

static binary instrumentation on the kernel image. Then, it adds a new section to the binary. This new section contains the memory reservation code. Later, KUP finds a specific function where the memory reservation code should run first. Then KUP hooks that function and injects the memory reservation function's address, so that this function can run before the specified function. After its execution, the code jumps back to the specified function and resumes the normal kernel execution.

To handle the page faults of the restored process, KUP relies on dynamic kernel instrumentation by hooking the kernel's page fault handling function. KUP executes its own page fault handling code, which binds the page corresponding to the stored virtual-to-physical mapping of the restored process.

Hence, by using both instrumentation techniques, we believe that KUP's PPP is easily applicable to commodity OSes while strictly adhering to the design goals of KUP.

PPP in action. KUP enables PPP with the following five concrete steps:

(1) Before checkpointing, KUP performs binary patching on the new kernel image.
(2) During checkpointing, the virtual-to-physical mapping of the targeted application is passed to a kernel module that reserves a memory space where the per-process page mapping information is saved.
(3) During new kernel's reboot, with the binary patched module, KUP globally reserves the set of per-process requested pages passed on from the previous kernel.
(4) Before application restoration, KUP patches the page fault handler with its own version to specially handle the page faults from the restored application.
(5) During restore, KUP only restores the states of the process except the memory pages and then restarts the application.

Steps (1), (2), and (4) implicitly hook the kernel. In step (1), we patch the new kernel image with our memory reservation code. In step (2), we use a kernel module for saving the per-process mapping information passed to it during the checkpoint phase. In step (4), KUP dynamically patches the page fault handler to specially consider the page faults from the restored process. After handling all the page faults, KUP falls back to the original function, thereby not imposing any overhead after the restoration.

By using PPP, KUP can dramatically reduce the total downtime with respect to the application C/R. Unlike dynamic hot-patching, PPP is applicable across multiple kernel updates, since it does not modify any in-kernel data structures. Instead PPP leverages the stable in-kernel functionality for the memory reservation and page fault handling. PPP does not introduce any security issues as the superuser is only responsible for the whole update procedure. Further, PPP's approach inherently adopts on-demand restore, as it naturally rebinds the faulted page to the virtual address, whereas FOAM explicitly binds the process's address to an image-file backed file descriptor to enable on-demand restore.

Component	Lines of code
criu / on-demand restore	810 lines of C
criu / FOAM	950 lines of C
criu / PPP	600 lines of C
KUP systemd, init	1040 lines of Python/Bash
criu / others, kexec(), etc.	150 lines of C
Total	3,550 lines of code

Table 1: An implementation complexity (lines of code) of KUP.

5.3 Safe Fallback

After updating the kernel, a system or an application may not run correctly for many different reasons. For example, an application may not run correctly due to the updated buggy file system [9] or a buggy system configuration [59]. In such cases, KUP's key ability can be retrofitted to support safe fallback, i.e., by carrying out instant downgrade. To allow the OS and applications to recover from failure, KUP takes exactly the same steps, but with the previous kernel image.

In particular, before switching kernels, KUP loads the safe fallback kernel image in a reversed memory section. Then, if the failure occurs, KUP performs the instant downgrade using this safe fallback kernel image. Depending on where such a fault occurs, this downgrade can be triggered because of a kernel layer or an application layer. KUP supports both cases: (1) if a fault is detected at the kernel layer (i.e., a kernel panic), KUP switches to the previous kernel; (2) if a fault is detected at the application layer (i.e., an application restoration fails), KUP switches into the previous kernel with the help of the system management daemon. Note that policies to detect these update failures are orthogonal to KUP's approach, as one can easily plugin any custom policy into KUP.

6 Implementation

We implemented our prototype of KUP on Linux v3.17 by using criu (v1.4) [22] for application C/R and used kexec() [31] for the in-place kernel switch. We modified various components of criu and implemented userspace binary patching module in python along with the pluggable kernel modules for PPP. The whole code consists of 3,550 lines in total (see Table 1).

FOAM. Besides application C/R, KUP also relies on criu for achieving quiescence, which is obtained as soon as the kernel pulls the process out of the sleeping state. criu uses the ptrace [37] functionality to achieve this. This ensures that KUP can only begin the application checkpointing when the process is back in the userspace as the kernel notifies the KUP process. Moreover, KUP concen-

Date	Ubuntu minor update From → To	Updatable? KUP	Updatable? kpatch	Analysis S	Analysis D	Patch complexity #files	Patch complexity +#lines	Patch complexity -#lines	#Bugs (security)	Example (a reason for failure)
2014/05/15	Linux 3.13.0-24 → 27	✓	-	18	16	150	1,346	1,480	139 (5)	New `.is_fw_header` in `rtl_has_ops` struct
2014/06/04	Linux 3.13.0-27 → 29	✓	-	4	0	178	1,495	984	168 (4)	New static var. `pirq_ite_set` in `pirqmap()`
2014/07/15	Linux 3.13.0-29 → 32	✓	-	20	14	551	11,890	5,694	756 (7)	New `.cache_events` in `power_pmu` struct
2015/08/13	Linux 3.13.0-32 → 34	✓	✓	0	0	6	77	18	10 (4)	CVE-2014-4943 is fixed.
2015/06/14	Linux 3.19.0-20 → 21	✓	✓	0	0	4	31	2	1 (1)	CVE-2015-1328 is fixed.

Table 2: Snippets of our analysis on minor updates spanning twelve months, in the Ubuntu distribution, using KUP and `kpatch` (refer Figure 1 for a full picture). Unlike `kpatch` failed to support 21 cases out of 23, KUP was able to support all 23 cases (checks under *Updatable?*). To understand why `kpatch` cannot support such updates, we analyzed each minor release to see (1) how many new *static variables* (*S* under *Analysis* column) are introduced, and (2) how many *data structure layout* (*D* under *Analysis*) changes are included. To be specific, we also included an example of such cases that `kpatch` cannot easily support, even with manually designed update payload.

trates on per-application C/R that only saves selected applications during an update. During restore, `criu` restarts the process with the help of parasite, thereby allowing the application to resume from where it was suspended after reconstructing all of its states.

To enable on-demand restoration, we `mmap()` process' virtual pages with the checkpointed image. For this, we modify the `criu`'s *parasite* code, an injected process coordinates process restoration, to correctly back the memory pages by using the proper offset from the image file.

We implement `FOAM` by modifying both the incremental checkpoint implementation and the basic restore implementation of `criu`. For `FOAM`, we use `xfs` as KUP's backend file system to store the checkpointed image as `xfs` supports a filesize up to 16 TB. Otherwise, this technique can be generally applied by serializing the virtual address spaces to multiple image files (see §5.1). KUP maps the entire virtual address space into file offsets and then creates *holes* by truncating it to 2^{48-1} bytes.

`PPP`. We implement `PPP` by hooking the memory subsystem via dynamic kernel instrumentation and static instrumentation. Before rebooting the newer kernel, we perform binary patching on the kernel binary (`vmlinuz`) by hooking the `setup_arch()` function for memory and page reservation for the restoring process. The `setup_arch()` function is one of the oldest functions existing in the kernel which we hook. While checkpointing, a kernel module obtains the virtual address-to-page frame number mapping from `criu` (via `ioctl`) and reserves the information in a global memory and marks the pages to be unusable. At the time of reboot, the hooked function reserves the process' pages and the global memory with the help of the boot allocator. Later, the page fault handler refers the global memory for accessing the relevant pages corresponding to the restoring process. During restore, KUP uses `jprobes` to achieve the on-demand page fault handling by hooking the `handle_mm_fault()`, which updates the page table entry of the restoring process after binding the page to the corresponding faulted address. We do not modify any in-kernel data-structures for `PPP`.

Optimizing `kexec()`. Unlike dynamic hot-patching techniques that employ unstable in-place updates (see Figure 1), KUP seeks a robust way to replace the entire kernel, thus allowing a complete switching between two different kernel versions. Instead of relying on hard or soft reboot, we use `kexec()`, which is originally designed to debug a crashed kernel with a secondary kernel image in a post-mortem manner. `kexec()` is robust as it supports all minor updates of Ubuntu from May 2014 to July 2015, and is over 50% faster than soft reboot (see Table 3). `kexec()` acts as a minimal bootloader. It is responsible for resetting device states before booting into a new kernel.

We make two optimizations to the stock `kexec()` to reduce the booting time: (1) avoid polling of `PCIe` slots, if they are not used before switching the kernel; (2) bring CPUs *lazily* online during boot, since we observed that the synchronization among system services slows down the system's boot.

7 Evaluation

We evaluate KUP by answering the following questions:

- How effectively can KUP apply patches compared to popular dynamic hot-patching techniques? (§7.1)
- How much downtime does KUP incur while running various types of applications during updates? (§7.2, §7.3)
- How effective is each technique in decreasing the system downtime during updates? (§7.4)
- How effective is KUP in a full software stack composed of multiple inter-related applications? (§7.5)

7.1 Dynamic Hot-patching vs. KUP

We compare KUP with `kpatch` (latest version—v0.2.2) to show its effectiveness when switching between various versions of Ubuntu kernel releases. Table 2 shows in-depth results on five specific patches out of 23 patch cases discussed in Figure 1. Overall, KUP is able to support all 23 updates, whereas `kpatch` fails to update 21 versions except two. It fails if any of the following happens in the patch: (1) build and diff failure for the kernel versions; (2)

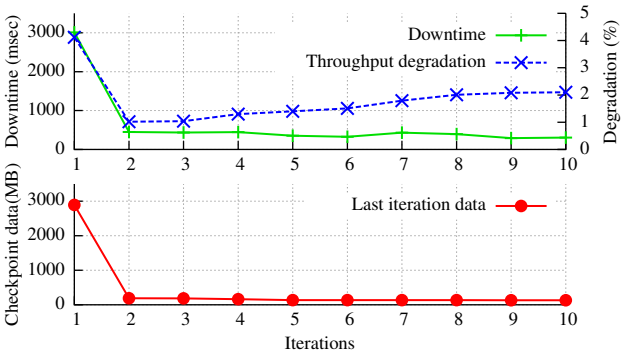


Figure 3: Throughput degradation, downtime and the data dumped in the last iteration measured for varying iterations during incremental checkpoint of `memcached`.

a patch modifies a value of static local variables (shown as *S* in *Analysis* column); (3) a patch changes the layout of a data structure (shown as *D* in *Analysis* column).

The first case, build and diff failure, is an implementation bug of `kpatch` and is not related to the inherent limitations of dynamic hot-patching techniques. We evaluated 23 patches and `kpatch` fails for 11 such cases. The second case consisting of the update of static local variables' values, causes a conflict with its runtime value semantics after hot-13 patches belong to this category that can be handled via kernel module. The last case entails a layout change in a data structure, in which `kpatch` has to locate and update all of its instances during runtime to safely update the system, but completely identifying them is far beyond the current state-of-the-art techniques and involves many challenging problems in the domain of program analysis [17]. `kpatch` fails on 11 patches for this case.

The aforementioned causes can be related to the patch complexity, which we measure in terms of the number of files and lines of code changed (*Patch complexity* column) and the number of bugs addressed (*#Bugs* column). The kernel updates, 3.13.0-32 → 34 and 3.19.0-20 → 21, are the only two updates that `kpatch` can handle, as there are relatively few changes in the code: the former changes 6 files and 71 lines of code to fix 10 bugs, and the latter one modifies 77 files and 31 lines of code to fix one bug. All other updates alter more than 150 files and 1346 lines of code, thereby resolving 139 bugs.

7.2 Applying KUP with Running Applications

In this section, we evaluate KUP with various types of real applications and provide our analysis of the impact of KUP on their performance across a system update. KUP is able to handle all major updates from v3.17-rc6 to v4.1. We perform all of our experiments on a 4 core machine with 64 GB RAM.

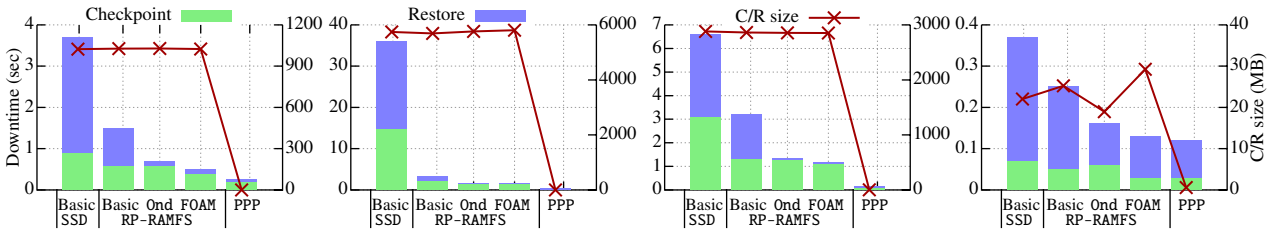
Faster storage medium: `RP-RAMFS`. As discussed in §5, the backend storage medium can inhibit the end-to-end performance of KUP. Users can circumvent this by adopting a RAM-based file system (reboot-persistent RAM file system—`RP-RAMFS` [18]) that stores its data purely in RAM but makes it persistent across system reboots. Note that `RP-RAMFS` is complementary to any C/R, even though it can dramatically decrease the application downtime. We use `RP-RAMFS` to show that users can use the emerging persistent memory [62] as a persistent-across reboot memory for the application C/R. However, `RP-RAMFS` has one fundamental limitation: the system needs enough free RAM to store the snapshot.

Targeted applications. While KUP can support most of applications (see §8 for limitations), we select four applications, namely `memtester` [10], `memcached` [42], `mysql` and the Linux kernel compilation (`LKC`), to show the effectiveness of KUP. `memtester` is a memory-write intensive application that is used for finding faults in RAM. `memcached` is a in-memory key/value store and we use `memaslap` as a client for load generation. `memaslap` runs on a different machine. `mysql` is a relational database. We use `linkbench` [3], which is a read-dominated benchmark that processes social graphs by using `mysql` as a database. We chose `memcached`, `memtester` and `linkbench` to illustrate memory-heavy workloads, and `LKC` to represent multi-process and computation-heavy program with small working set size (around 25 MB).

Incremental checkpoint iterations. In regard to `FOAM`, KUP's only tunable parameter is the number of iterations used for the incremental checkpoint. To see how the iteration count affects system behavior, we measure throughput, downtime, and amount of data written in the last iteration for `memcached` by varying the iteration count. Even for memory-heavy `memcached`, Figure 3 shows that there were no meaningful differences in both throughput (1-2%) and downtime since the incremental checkpoint quickly converges from the second iteration. This is because, unlike previous studies based on slow network-based incremental checkpointing mechanisms [12, 64], KUP uses fast local storage or memory for checkpointing, making iterations non-critical. For all other experiments in this paper, we set the iteration count to two.

Results. KUP can successfully update the Linux kernel from v4.0 to v4.1 while running the aforementioned programs. Figure 4 shows the exact downtime and the data checkpointed for various schemes of KUP. `PPP` outperforms all other techniques as it only saves the page frame numbers (corresponding to the actual pages' physical address) rather than the actual page contents, thereby decreasing the checkpointed size by orders of magnitude. However, there is not much difference between `PPP` and `FOAM` for `LKC`, because of its smaller working set size.

`FOAM` efficiently supports both incremental checkpointing and on-demand restore. Because of the incremental checkpoint, only the data that gets checkpointed in the



(a) `memtester` **(b)** `memcached` **(c)** `mysql` **(d)** Linux kernel compile

Figure 4: Downtime breakdown of checkpoint-and-restart on various applications when updating a kernel from 4.0 to 4.1 with KUP.

last iteration contributes to the observable downtime. Evidently, due to KUP's simple data structure used for image checkpointing, `FOAM`'s image size remains the same as that of the basic approach. Figure 4 shows negligible downtime for on-demand restore, thus proving the advantage of using on-demand restore for all three approaches: single checkpointed data (marked `Ond`), `FOAM` and `PPP`, against basic restore.

7.3 End-to-end Analysis of KUP's Approach

In this section, we cover an end-to-end performance evaluation of a web service, `memcached` (approximately consuming 5.6 GB with 20% percent write operations), while updating the system's kernel from v4.0 to v4.1. We use `bwm-ng` tool [76] to understand its downtime during update by end-users. Since `memcached` is a memory intensive service caching large amounts of data, where data loss is critical to performance after system update, it fits well for KUP's purpose and effectively shows the impact of our techniques.

Effectiveness of techniques. Figure 5 exhibits the impact of KUP on bandwidth utilization by the client performing `get`/`set` operations on `memcached` server during updates. `PPP` shows the best performance (refer (i)) in terms of shortest downtime and least performance degradation (only 0.68% over the period of 300 seconds) after being restored. Since, `PPP` instantly shifts to `kexec()`, this can be beneficial to applications that are either time critical or that execute for shorter durations. Not only that, `PPP` is storage -independent thanks to its efficient memory mapping storage (see §7.4).

In terms of downtime, `FOAM` performs the same as `PPP` (refer (h) and (i)). This is because `FOAM` provides the best of both worlds by combining both state-of-the-art techniques and has least downtime compared to the individual techniques ((d) and (h)). However, one disadvantage of `FOAM` is that it disrupts the bandwidth during incremental checkpoint (starts around 192 second in (h)), even though it keeps server alive.

The basic technique ((a) and (e)) suffers the most, whereas the incremental checkpoint ((b) and (f)) and on-demand restore ((c) and (g)) techniques perform almost the same, as both reduce their respective downtime in two

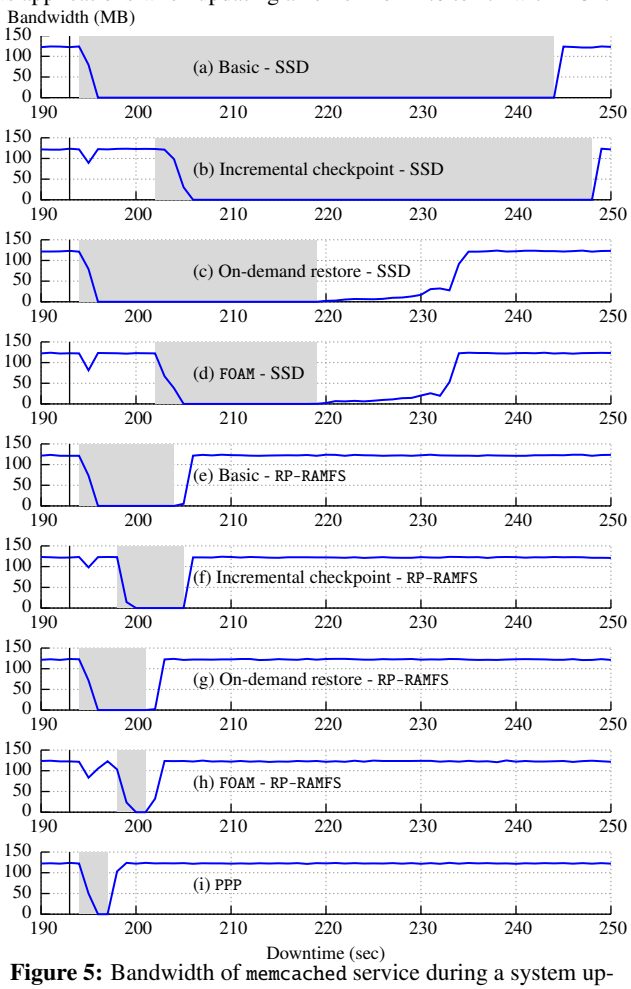


Figure 5: Bandwidth of `memcached` service during a system update. We measured the performance of `memcached` and initiated the kernel update with KUP (a bar near 192 second). From top to bottom, each graph demonstrates properties of each technique, baseline, incremental checkpoint, on-demand restore, and `FOAM` on either SSD or `RP-RAMFS` as a backend medium, and lastly `PPP`. The boxes represent the time from which the network throughput starts decreasing due to the in-progress update.

different steps of OS update. Besides this, the on-demand restore helps the `memcached` server to instantly activate—the bandwidth utilization is non-zero instantly after the OS update (refer (c), (d), (g) and (h)). KUP can further leverage Halite's prefetching approach [81] to improve performance during on-demand restore of `memcached` ((c)

Machine	Soft reboot	kexec() reboot	
	Default	Default	Optimized
1-way 4-core E5-1620/ 64 GB	45.2	2.4	2.4
1-way 8-core E3-1271/ 32 GB	42.9	6.7	6.0
2-way 16-core E5-2630/128 GB	62.9	22.1	9.2
8-way 80-core E7-8870/512 GB	247.5	31.9	25.9

Table 3: Kernel switch time in seconds for different hardware configurations. We used a 1-way 4-core machine for evaluation.

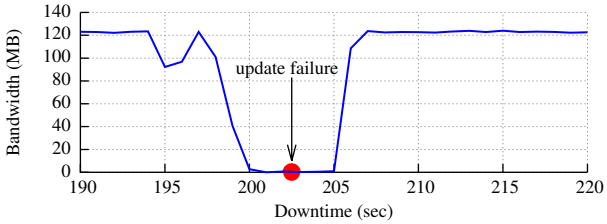


Figure 6: We simulated the update failure by triggering a fault right after system's boot (red mark). KUP's safe fallback mechanism allows memcached to continue running on the previous kernel even after the update failure.

and (d)). Likewise, the incremental checkpoint (refer (b) and (f)) also decreases the downtime during when compared with the basic approach.

Downtime from kernel switch. Although the kernel switch has no effect on application C/R, it adds significant overhead as downtime. Furthermore, this downtime varies with various hardware configurations. Table 3 shows both of the soft and kexec() reboot times. The reboot time is the time taken to boot into a new kernel from the previous one. For machines with increasing core count, both of the soft reboot time as well as the default kexec() time increase with increasing hardware complexity. On further analysis, we found that most of the time is spent in purgatory [31], which runs in between two kernels and initializes hardware components such as PIC and VGA. This holds true even for the bigger machines (>16 cores), which we need to investigate further. In addition, the new kernel also spends time initializing the system services such as network services and filesystem mounting. We reduce the system boot time (refer Table 3) at two places during the new kernel boot: (1) by lazily bringing up each core, which saves six seconds for the 80-core machine, and (2) by skipping the polling of unused PCI slots, thereby saving 8.5 seconds on the 16-core machine.

Safe fallback upon a failure. KUP inherently provides safe fallback mechanism by using kexec(). To show its effectiveness, we intentionally inject a fault in the restore process right after the new kernel boots up. As soon as KUP detects a fault, it initiates a downgrade to its previous kernel image, which we stored at the initiation of the update. Figure 6 shows the performance of memcached upon such an event. As expected, the downtime of memcached becomes approximately two times longer (five seconds) than KUP's update without an error. The safe fallback approach is only used once we are confident we can safely restart the application on the older kernel. We further test the KUP's safe fallback approach with a buggy file system [9] while running Linux kernel compile. Since KUP could not resume compilation in the updated kernel, it reverted to the previous version.

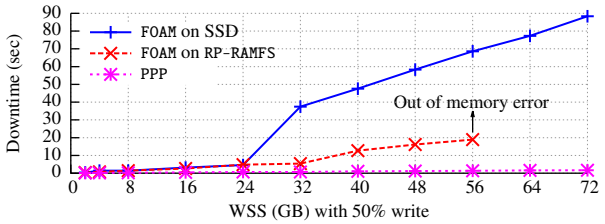


Figure 7: Performance of PPP and FOAM (on RP-RAMFS and SSD) with varying WSS (with 50% write) up to 72 GB, which is larger than a half of system's memory, 128 GB. PPP efficiently supports applications with large WSS, whereas RP-RAMFS fails as it requires free RAM space for checkpointing.

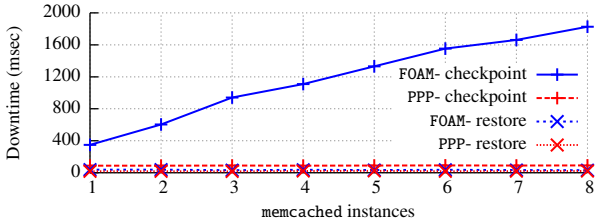
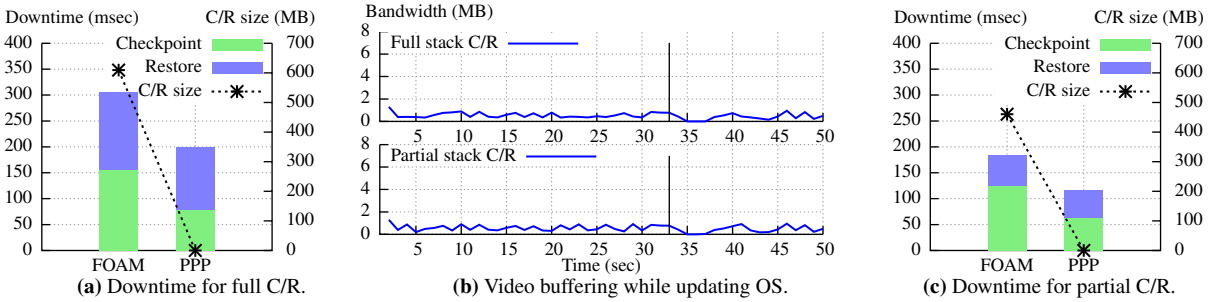


Figure 8: Downtime during checkpoint and restore phase for both FOAM and PPP approaches for multiple instances of memcached on SSD.

7.4 Micro-benchmarking FOAM and PPP

In this section, we discuss the impact of PPP and FOAM for various storage media by performing two experiments. This first of these is a microbenchmark-based evaluation which measures the downtime when checkpointing a process with varying working set size (up to 72 GB) with 50% write (Figure 7). The micro-benchmark allocates a certain amount of memory and endlessly dirties the pre-specified part of the allocated memory. By varying the working set size (WSS) from 1 GB to 72 GB on both SSD and RP-RAMFS, we measure the downtime incurred by both techniques. Figure 7 shows the effectiveness of using PPP as it outperforms all the techniques by 9.5-98.3× for SSD and 3.9-14.1× against RP-RAMFS by avoiding redundant copies of process's memory. KUP cannot rely on RP-RAMFS as it fails to work beyond 56 GB and 128 GB is the machine's total memory size.

On the other hand, the second experiment measures the downtime incurred while checkpointing and restarting multiple memcached instances (one to eight) on SSD while updating the OS with KUP (Figure 8). The experiment illustrates that (1) KUP is an effective per-application -based C/R mechanism that can checkpoint multiple applications in parallel and (2) PPP is the best approach for updating an OS with multiple applications are run-



(a) Downtime for full C/R. **(b)** Video buffering while updating OS. **(c)** Downtime for partial C/R.

Figure 9: Updating OS with a full-stack media streaming service—`ampache`. (a) shows the downtime observed when performing C/R on `ampache` with KUP. (b) shows the video buffering observed on the client side during OS update. Full web-stack C/R corresponds to updating the complete stack, whereas partial web-stack C/R is the careful C/R performed by the system administrator while updating the OS with KUP.

ning. Figure 8 shows the benefit of using `PPP` against `FOAM`. `FOAM`'s checkpointed data determines its downtime. Thus, with increasing `memcached` instances, KUP's overall downtime increases linearly. However, since `PPP` does not require I/O operations for checkpointing, its downtime remains constant regardless of the number of running instances.

7.5 Supporting the Full Software Stack

Here, we discuss the flexibility of KUP in updating a full software stack made up of typical co-running applications on general purpose servers inside data centers. We use `ampache` as a full software stack. We try to update it with and without the stack's knowledge. `ampache` is a `php` based media streaming service running with `mysql` to store the information about the users and songs. We modify `ampache` to also store movies in `mysql`. Figure 9 shows the downtime incurred because of C/R and as well as the buffering observed on the client side in terms of packets received while streaming a movie clip. On the server side, KUP checkpoints both `apache` and `mysql` and the maximum downtime observed because of C/R is around 300 and 200 milliseconds for `FOAM` and `PPP` respectively (refer Figure 9(a)). On the client side, the buffering stops for around three seconds and then resumes. However, the user does not observe any interruption while watching the video.

Since KUP is a per-application -based OS update mechanism, the administrator can perform partial web-stack C/R during the update by updating only a particular component of the whole service, assuming the administrator has a complete understanding of the full software stack. For example, in our case, KUP should only checkpoint `mysql` and leave the `apache` to the system management daemon. With this approach, the downtime can be further reduced by approximately 100 milliseconds (refer Figure 9(c)). Even though there is no observable improvement on the client side, such selective partial C/R can further reduce downtime in complex software stacks. Moreover, we confirm that the video and song keep on streaming even after the update, which helps us in verifying the correctness of our system.

8 Discussion

In this section, we discuss the limitations of KUP's approaches of using application C/R for instant kernel updates, and we discuss potential optimizations to further reduce the downtime imposed by the kernel switch.

Limitations. KUP inherits the limitations of `criu` for application C/R, such as limited support of `compat` mode of x86-64. This restricts the type of application that KUP can support. Furthermore, application C/R schemes find it difficult to restore the *external resources*. However, in the context of updating OSes, there are many opportunities to overcome these limitations. Because of the shorter time gap between checkpoint and restart in KUP, the host's external interfaces can easily resolve such uncertainty by describing some forms of buffering mechanisms. For example, a `TCP/IP` state will be preserved for about 75 seconds (default in Linux [29]) until the reset packet is fired. As long as the effective downtime is reasonably small, KUP can optimistically restore the checkpointed applications correctly, as in `memcached` §7.3 and full software stack §7.5. Currently, there is an ongoing effort to reliably preserve the TCP/IP states across updates or migrations [16]. Besides this, the current implementation of KUP partially supports unix domain sockets and desktop applications. Another point is that the downtime due to KUP's `FOAM` is still sensitive to the number of cache-to-disk writes. In the future, we would like to explore mechanisms that keep dirty pages across updates.

Using different kernel versions may affect the restoration process, thus adding another limitation to KUP usage. If the new kernel (or old kernel on downgrade) hampers its backward (or forward, respectively) compatibility (e.g., dropping a support of a certain system call, or even changing the interface of certain system calls), the restoration might fail, but we believe this is rare (or unlikely) in practice, as surveyed in §7.1. If the new kernels handle the application-specific states differently, this may result in latent application corruption. However, we are yet

to encounter such unexpected behavior. Another limitation can be that the newer kernel looses the old kernel data. This holds true only for applications that KUP do not checkpoint. Besides that, the kernel's memory state is independent of the checkpointed applications, thereby making them worthless for the checkpoint.

Suitable applications. We believe that KUP is suitable for all types of applications. By using PPP approach, KUP can update any type of application ranging from memory-intensive to I/O-intensive workloads. However, if there is any memory management modification or a page corruption occurs after the update, then the safe fallback technique will also fail. On the other hand, FOAM technique is not a suitable candidate for write-intensive workloads that frequently allocate and free memory pages. However, it provides the guarantee of safely falling back to the previous kernel with more confidence as the process' data is persisted on the disk. Therefore, the system administrators should choose wisely, depending on the criticality of the application and its service.

Optimizing reboot. The major source of downtime in KUP is the in-place kernel switch, depending on `kexec()`; varying from 2.4 sec to 25.9 sec based on the underlying hardware. Although we applied a few basic optimizations to `kexec()` (§6), we can further improve it by adopting a number of promising techniques demonstrated by previous research. For example, Nooks [74] develops shadow drivers techniques that preserve its internal states upon failure, and Live Update [75] uses a similar technique to share its internal state across device driver updates. Another promising direction is to carry over device driver states across reboot as there will be no hardware changes in the middle of system updates.

9 Conclusion

KUP is a simple yet robust update mechanism that instantly updates a running kernel across major kernel versions. KUP checkpoints user applications, then replaces the existing kernel with an updated kernel (any kernel version), and finally restores the checkpointed applications thereafter. KUP achieves its goal of instant kernel update by efficiently merging both checkpoint and restore techniques with the help of an optimized checkpoint format. Moreover, we come up with a complementary memory persistence mechanism that solves the inherent problem of userspace C/R and further improves KUP's performance while catering to disk-less systems. We believe that KUP is the first work that realizes swift kernel updates without modifying any kernel source. This makes KUP robust enough to be used in practice, thus allowing users to enjoy instant system updates for security patches, bug fixes and performance improvements with minimal disruption.

10 Acknowledgment

We thank the anonymous reviewers at ATC'16, and our shepherd, Hani Jamjoom, for their helpful feedback, as well as the operations staff for their proofreading efforts. This research was supported by the NSF award DGE-1500084, ONR under grant N000141512162, DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006, ETRI MSIP/IITP[B0101-15-0644], and NRF BSRP/MOE[2015R1A6A3A03019983].

References

[1] ANDERSON, G., AND RATHKE, J. Dynamic Software Update for Message Passing Programs. In *APLAS* (2012), Lecture Notes in Computer Science, Springer, pp. 207–222.

[2] ANSEL, J., ARYA, K., AND COOPERMAN, G. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium* (Rome, Italy, May 2009).

[3] ARMSTRONG, T. G., PONNEKANTI, V., BORTHAKUR, D., AND CALLAGHAN, M. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13, pp. 1185–1196.

[4] ARNOLD, J., AND KAASHOEK, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the ACM EuroSys Conference* (Nuremberg, Germany, Mar. 2009), pp. 187–198.

[5] BAUMANN, A., APPAVOO, J., WISNIEWSKI, R. W., SILVA, D. D., KRIEGER, O., AND HEISER, G. Reboots Are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the 2007 ATC Annual Technical Conference (ATC)* (Santa Clara, CA, June 2007), pp. 26:1–26:14.

[6] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (Big Sky, MT, Oct. 2009), pp. 29–44.

[7] BAUMANN, A., HEISER, G., APPAVOO, J., DA SILVA, D., KRIEGER, O., WISNIEWSKI, R. W., AND KERR, J. Providing Dynamic Update in an Operating System. In *Proceedings of the 2005 ATC Annual Technical Conference (ATC)* (Anaheim, CA, Apr. 2005), pp. 279–291.

[8] BOYD, E. Ext4 corruption associated with shutdown of Ubuntu 12.10, 2012. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/1073433.

[9] BRENTAAR. btrfs forced readonly, 2014. https://bbs.archlinux.org/viewtopic.php?id=188900.

[10] CAZABON, C. memtester(8) - Linux man page, 2009. http://linux.die.net/man/8/memtester/.

[11] CHEN, H., CHEN, R., ZHANG, F., ZANG, B., AND YEW, P.-C. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)* (2006), pp. 35–44.

[12] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005), pp. 273–286.

[13] CLEMENTS, A. T., KAASHOEK, M. F., AND ZELDOVICH, N. RadixVM: Scalable address spaces for multithreaded applications. In *Proceedings of the ACM EuroSys Conference* (Prague, Czech Republic, Apr. 2013), pp. 211–224.

[14] COMMUNITY. OpenVZ Linux Container, 2005. http://openvz.org.

[15] CORBET, J. Preparing for user-space checkpoint/restore, 2012. https://lwn.net/Articles/478111/.

[16] CORBET, J. TCP connection repair, 2012. https://lwn.net/Articles/495304/.

[17] CUI, W., PEINADO, M., XU, Z., AND CHAN, E. Tracking Rootkit Footprints with a Practical Memory Analysis System. In *Proceedings of the 21st Usenix Security Symposium (Security)* (Bellevue, WA, Aug. 2012), pp. 42–56.

[18] DAVYDOV, V. pram: persistent over-kexec memory file system, 2013. https://lwn.net/Articles/561330/.

[19] DEPOUTOVITCH, A., AND STUMM, M. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the ACM EuroSys Conference* (Paris, France, Apr. 2010), pp. 181–194.

[20] DUMITRAŞ, T., AND NARASIMHAN, P. Why Do Upgrades Fail and What Can We Do About It?: Toward Dependable, Online Upgrades in Enterprise System. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware '09)* (New York, NY, USA, 2009), Springer-Verlag New York, Inc., pp. 18:1–18:20.

[21] DUMITRAS, T., NARASIMHAN, P., AND TILEVICH, E. To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains. In *ACM SPLASH Onward* (Oct. 2010), pp. 865–876.

[22] EMELYANOV, P. CRIU: Checkpoint/Restore In Userspace, July 2011. http://criu.org/Main_Page.

[23] ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. W. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, CO, Dec. 1995).

[24] FACEBOOK. Open Compute Project, 2015. http://www.opencompute.org/.

[25] FISCHETTI, T. Why is my OS X Yosemite install taking so long?: an analysis, Oct. 2014. http://www.r-bloggers.com/why-is-my-os-x-yosemite-install-taking-so-long-an-analysis.

[26] GHOSHAL, D., RAMKUMAR, S. R., AND CHAUHAN, A. Distributed Speculative Parallelization using Checkpoint Restart. In *ICCS* (2011), Procedia Computer Science, Elsevier, pp. 422–431.

[27] GIUFFRIDA, C., IORGULESCU, C., AND TANENBAUM, A. S. Mutable Checkpoint-restart: Automating Live Update for Generic Server Programs. In *Proceedings of the 15th International Middleware Conference* (2014), Middleware '14, pp. 133–144.

[28] GIUFFRIDA, C., KUIJSTEN, A., AND TANENBAUM, A. S. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013), pp. 279–292.

[29] GODOY, J., HOGBIN, E., KOMARINSKI, M., AND MERRILL, D. TCP keepalive time in Linux, 2015. http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/usingkeepalive.html.

[30] GOEL, A., CHOPRA, B., GEREA, C., MÁTÁNI, D., METZLER, J., UL HAQ, F., AND WIENER, J. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Utah, USA, June 2014), pp. 541–549.

[31] GOYAL, V. kexec: A new system call to allow in kernel loading, 2014. https://lwn.net/Articles/582711.

[32] HALLYN, S., AND GRABER, S. LXC - Linux Containers, 2013. https://linuxcontainers.org.

[33] HARGROVE, P. H., AND DUELL, J. C. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In *Journal of Physics: Conference Series* (2006), p. 494.

[34] HAYDEN, C. M., MAGILL, S., HICKS, M., FOSTER, N., AND FOSTER, J. S. Specifying and Verifying the Correctness of Dynamic Software Updates. In *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments* (2012), VSTTE'12, pp. 278–293.

[35] HAYDEN, C. M., SMITH, E. K., DENCHEV, M., HICKS, M., AND FOSTER, J. S. Kitsune: Efficient, General-purpose Dynamic Software Updating for C. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2012), OOPSLA '12, pp. 249–264.

[36] HAYDEN, C. M., SMITH, E. K., HICKS, M., AND FOSTER, J. S. State transfer for clear and efficient runtime updates. In *ICDE Workshops* (2011), IEEE.

[37] HEO, T. ptrace: implement PTRACE_SEIZE/INTERRUPT and group stop notification, 2011. https://lwn.net/Articles/441990/.

[38] HINES, M. R., DESHPANDE, U., AND GOPALAN, K. Post-copy Live Migration of Virtual Machines. *ACM SIGOPS Operating System Review*, 3 (July 2009), 14–26.

[39] HOSEK, P., AND CADAR, C. Safe Software Updates via Multi-version Execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)* (2013), pp. 612–621.

[40] HUANG, Y., KINTALA, C., KOLETTIS, N., AND FULTON, N. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on* (June 1995), pp. 381–390.

[41] HYKES, S. Docker: Build, Ship and Run Any App, Anywhere, 2013. https://www.docker.com.

[42] INTERACTIVE, D. Memcached: Distributed memory object caching system, 2003. http://memcached.org/.

[43] JANAKIRAMAN, G. J., SANTOS, J. R., SUBHRAVETI, D., AND TURNER, Y. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2005), 260–269.

[44] KELLY, G. Windows 10 Automatic Updates Start Causing Problems, 2015. http://www.forbes.com/sites/gordonkelly/2015/07/25/windows-10-automatic-update-problems.

[45] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Efficient patch-based auditing for web application vulnerabilities. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012).

[46] KIM, T., CHANDRA, R., AND ZELDOVICH, N. Recovering from intrusions in distributed systems with Dare. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)* (Seoul, South Korea, July 2012).

[47] KIM, T., WANG, X., ZELDOVICH, N., AND KAASHOEK, M. F. Intrusion recovery using selective re-execution. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)* (Vancouver, Canada, Oct. 2010).

[48] KIM, T., AND ZELDOVICH, N. Practical and effective sandboxing for non-root users. In *Proceedings of the 2013 ATC Annual Technical Conference (ATC)* (San Jose, CA, June 2013).

[49] LAADAN, O., AND NIEH, J. Transparent Checkpoint-restart of Multiple Processes on Commodity Operating Systems. In *Proceedings of the 2007 ATC Annual Technical Conference (ATC)* (Santa Clara, CA, June 2007), pp. 25:1–25:14.

[50] LAWALL, J., AND MULLER, G. Efficient incremental checkpointing of Java programs. In *Proceedings International Conference on Dependable Systems and Networks (DSN)* (2000), pp. 61–70.

[51] LIM, G. Faster Booting in Consumer Electronics. Samsung Electronics, Industry Talks, ATC 2015.

[52] LOWELL, D. E., SAITO, Y., AND SAMBERG, E. J. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Boston, MA, Oct. 2004), pp. 211–223.

[53] MAKRIS, K., AND BAZZI, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 ATC Annual Technical Conference (ATC)* (San Diego, CA, June 2009).

[54] NEAMTIU, I., AND HICKS, M. Safe and Timely Updates to Multi-threaded Programs. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland, June 2009), pp. 13–24.

[55] NEAMTIU, I., HICKS, M., FOSTER, J. S., AND PRATIKAKIS, P. Contextual Effects for Version-Consistent Dynamic Software Updating and Safe Concurrent Programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (San Francisco, USA, January 2008), pp. 37–49.

[56] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical Dynamic Software Updating for C. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2006), pp. 72–83.

[57] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, Apr. 2013), pp. 385–398.

[58] NOACK, M. Comparative Evaluation of Process Migration Algorithms. 1–51.

[59] NVIDIA drivers not working after upgrade. Why can I only see terminal? http://askubuntu.com/questions/37590/nvidia-drivers-not-working-after-upgrade-why-can-i-only-see-terminal.

[60] ORACLE. Database rolling upgrade using Data Guard SQL Apply. Maximum Availability Architecture White Paper, May 2010.

[61] OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), pp. 361–376.

[62] PELLEY, S., CHEN, P. M., AND WENISCH, T. F. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (Piscataway, NJ, USA, 2014), ISCA '14, pp. 265–276.

[63] PINA, L., AND HICKS, M. Rubah: Efficient, General-purpose Dynamic Software Updating for Java. In *HotSWUp* (2013).

[64] PLANK, J. S., BECK, M., KINGSLEY, G., AND LI, K. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings* (Berkeley, CA, USA, 1995), TCON'95, pp. 18–18.

[65] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Newport Beach, CA, Mar. 2011), pp. 291–304.

[66] POTTER, S., AND NIEH, J. Reducing Downtime Due to System Maintenance and Upgrades. In *Proceedings of the 19th Conference on Large Installation System Administration Conference (LISA)* (2005), pp. 1–16.

[67] REDHAT. kpatch: dynamic kernel patching, 2014. https://github.com/dynup/kpatch.

[68] SALUSTE, M. Repair or reinstall Windows Update. https://www.winhelp.us/reinstall-windows-update.html.

[69] SINIAVINE, M., AND GOEL, A. Seamless Kernel Updates. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (Washington, DC, USA, 2013), pp. 1–12.

[70] SOULES, C. A. N., SILVA, D. D., AUSLANDER, M., GANGER, G. R., AND OSTROWSKI, M. System Support for Online Reconfiguration. In *Proceedings of the 2003 ATC Annual Technical Conference (ATC)* (San Antonio, TX, June 2003), pp. 141–154.

[71] SPECTOR, L. When Windows Update won't update. http://www.pcworld.com/article/2098429/when-windows-update-wont-update.html.

[72] STOYLE, G., HICKS, M., BIERMAN, G., SEWELL, P., AND NEAMTIU, I. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, 4 (Aug. 2007).

[73] SUSE. Live Kernel Patching with kGraft, 2014. https://www.suse.com/promo/kgraft.html.

[74] SWIFT, M. M., BERSHAD, B. N., AND LEVY, H. M. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 4 (Nov. 2004).

[75] SWIFT, M. M., MARTIN-GUILLEREZ, D., BERSHAD, B. N., LEVY, H. M., SWIFT, M. M., MARTIN-GUILLEREZ, D., BERSHAD, B. N., AND LEVY, H. M. Live Update for Device Drivers. Tech. Rep. Technical Report CS-TR-2008-1634, University of Wisconsin Computer Sciences, Mar. 2008.

[76] VGROPP. bwm-ng, 2004. http://linux.die.net/man/1/bwm-ng/.

[77] WHALEY, J. System Checkpointing Using Reflection and Program Analysis. In *Proceedings of Reflection 2001, the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (Kyoto, Japan, Sept. 2001), A. Yonezawa and S. Matsuoka, Eds., LNCS, Springer-Verlag, pp. 44–51.

[78] YAMADA, H., AND KONO, K. Traveling Forward in Time to Newer Operating Systems Using ShadowReboot. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2013), VEE '13, pp. 121–130.

[79] ZARRABI, A., SAMSUDIN, K., AND WAN ADNAN, W. A. Linux Support for Fast Transparent General Purpose Checkpoint/Restart of Multithreaded Processes in Loadable Kernel Module. *Journal Grid Computing*, 2 (2013), 187–210.

[80] ZELLWEGER, G., GERBER, S., KOURTIS, K., AND ROSCOE, T. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, Colorado, Oct. 2014), pp. 17–31.

[81] ZHANG, I., DENNISTON, T., BASKAKOV, Y., AND GARTHWAITE, A. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proceedings of the 2013 ATC Annual Technical Conference (ATC)* (San Jose, CA, June 2013), pp. 1–12.

Apps with Hardware: Enabling Run-time Architectural Customization in Smart Phones

Michael Coughlin, Ali Ismail, Eric Keller
University of Colorado, Boulder

Abstract

In this paper we present a novel system which incorporates programmable hardware (an FPGA) into a smart phone to enable a vision where apps can include both software and hardware components, or apps with hardware. We introduce a novel mechanism to enable sharing the FPGA in a practical manner by leveraging the unique deployment model of mobile applications - namely that deployment is via an app store, where we introduce a new cloud-based compilation. We present our prototype smart phone using the Zedboard, which pairs a Xilinx Zynq FPGA with an embedded Cortex A9, running an Android-based system which we extended to provide run-time system support for dynamically managing apps with hardware and providing a secure loading system. With this prototype, our evaluation demonstrates the performance gains for an AES encryption module (representing cryptography), a QAM modulation module (representing software-defined radio) of 3x to several orders of magnitude, with room for improvement and a hardware-based memory scanner (representing custom co-processors). We demonstrate the feasibility of our cloud-based compilation within the context of real app store statistics. Finally, we present a case study of a complete integration of hardware into an existing application (the Orbot Tor client).

1 Introduction

In designing new smart phone devices, the vendor must operate under a number of constraints – form factor, functionality, cost, energy use, etc. This leads to the vendor making a number of decisions regarding the various tradeoffs. These decisions, however, can then lead to the case where the device has both too little (the application developers/users want more) and too much (the application developers/users don't use what is there). *What if there was a way to put these trade-offs into the hands of users and application developers?*

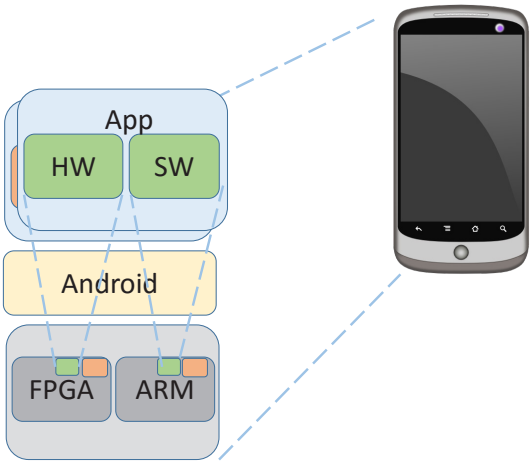


Figure 1: Smart phone with a processor (ARM) coupled with programmable hardware (FPGA).

In this paper, we present our 'apps with hardware' vision (illustrated in Figure 1), design, and implementation which incorporates programmable hardware, such as an FPGA (field programmable gate array), into a smart phone[1], and extends a mobile operating system to allow for application control of the current hardware configuration (*e.g.,* by including the hardware configuration with their app). The high-level idea is to couple software-like (re)programmability with hardware-like performance. In providing programmability, the phone vendor empowers the application developers (and by extension the end users) with the ability to influence the design decisions.

Developers, for example, would be able to introduce

[1]We envision this as being commercially available smart phones, not just in prototyping devices – a vision supported by the commercial availability of system-on-chip devices which already couple an ARM processor that is widely used in smart phones (such as the ARM Cortex A9 processor found in the iPhone 4) with reconfigurable logic [5, 25], with or more recently the ARM Cortex A53 [26], and further supported by recent advances by vendors where hardware modules can be designed using a high-level language, such as C++ [23].

(and deploy) new communication technologies, such as those that work on the emerging dynamic spectrum access paradigm [30], where they can perform 'software' radio at the needed hardware performance levels and gain system wide benefits (*e.g.,* from not needing phones to include many dedicated radio interfaces). Developers will also be able to introduce new accelerators, such as for cryptography or other parallel processing that improve overall performance and efficiency. Finally, developers will be able to introduce independent co-processors, which can, for example, provide additional security [57] capabilities not possible in today's smart phones. In general, we introduce programmability of the smart phone hardware by creating an architecture centered on an FPGA with an embedded processor – with which, as we've seen with other programmable technology, such as graphical processor units (GPUs) and FPGAs in other contexts within the network systems community, developers will find creative ways to use the available processing power [52, 59, 45, 41].

Previous research has proposed adding reconfigurability to mobile devices, such as [11] [61] [32], but they have limitations that prevent them from being used today, such as lacking a method to share FPGA hardware, a distribution system for applications, or integration into modern operating systems or devices. Therefore, there are a number of challenges that we need to address to make our vision possible. First, we need to be able to share an FPGA between different smart phone applications – existing FPGA hardware and software are heavily centered on running a single application and not on idea of temporal or spacial sharing of resources. Second, we need a way to distribute apps with hardware to smart phones with compatible hardware – there is no binary compatibility in FPGAs or operating system which abstracts resources. Finally, we need the ability to manage the FPGA so that applications only have access to authorized resources – while processors have been adapted overtime to isolate running tasks, FPGAs have not.

In this paper we present our Cloud RTR system that addresses these challenges. In doing so, we introduce a system-level contribution that makes use of cloud technologies and builds on existing FPGA technology that together solve a problem that has eluded researchers for years. Specifically, we make the following contributions:

A slot based solution that allows for practical FPGA sharing: A central need to be able to allow apps to span software and the FPGA hardware is to enable the FPGA to be shared, as apps will be concurrently running. Our approach is based on Run-time reconfiguration (RTR), or the ability to change an FPGA's configuration at runtime. Specifically, our Cloud RTR system builds on the idea of "slots" [38] [49] [42], or areas of the FPGA that can be reconfigured separately and shared between applications. To make this practical, where previous systems have failed, we provide a new approach to slot-based reconfiguration using a compilation system that abstracts away the underlying FPGA requirements. The resulting platform supports the use of slots at run-time, whereas previous systems only support slots at design time, and can share the FPGA between multiple parties, as we discuss next. Further, we introduce operating system services to manage slots at run-time to allow for on demand access from apps.

An app store based approach that allows for multiple parties to distribute apps: Without operating system and binary compatibility, envisioning a system which allows for multiple parties to create apps and have them be distributed to a wide variety of devices may seem difficult. We introduce a new app-store system which extends existing app stores to to allow for both the compilation and the distribution of apps with hardware. Developers can upload apps with hardware to an extended app store, which will interface with the compilation system in order to generate the required slot configurations. We extend the app store system further to ensure that these configurations are distributed to the correct devices in packaged apps, and we provide corresponding operating system support in order to install them.

A security manager that enforces access control to sensitive resources Our Cloud RTR architecture also provides a secure loading subsystem that ensures that only trusted applications have access to sensitive resources. This subsystem interfaces with the slot run-time management system to only allow for signed app hardware to access these resources. This system also uses common hardware security technology to ensure that applications cannot exploit the operating system to override these security restrictions.

In addition to the above system-level advances which enable the apps with hardware vision, we make the following contributions which evaluate and demonstrate their use:

Evaluation of the computational requirements of Cloud RTR: While our approach of performing some compilation in the cloud is, to some degree, simplistic, the fact that it has not been done before does, we feel, point to its novelty. Importantly, we go beyond simply proposing to compile in the cloud and extend our work to fully evaluate the computation requirements of such an app store to support this using data about the current app market ecosystem. We show that for compilation throughput per machine ranges from 51 to 121 apps per day, which translates to needing 1020 servers to support an app ecosystem where 1 percent of all apps use the reconfigurable logic for a case where there is 1000 phone variants.

Demonstration and evaluation of three applications:

In Section 2, we describe three example categories of applications that will benefit from an apps with hardware. For each, we implemented and evaluated a representative application (Section 6). Our evaluation of an app which offloads to a hardware based QAM module (a representative software-defined radio application) shows a 40x speedup and a hardware based AES module (a representative cryptography application) shows a 3x speedup (including all of the interface between hardware and software). Additionally, our evaluation of a simple memory security scanner (a representative architectural enhancement) that is capable of searching the entire system address space only results in 3% overhead for other software running. Finally, to understand the considerations when integrating into existing, complex, applications, we modified the open source and widely used Orbot [17] Tor [37] client for Android to include and use a hardware cryptography module (Section 7).

In the remainder of the paper we will further motivate the proposal to incorporate an FPGA into a smart phone (Section 2), describe past FPGA sharing attempts (Section 3), and describe our system, including our Cloud RTR architecture, which includes both the architecture for our slot compilation and app store extensions (Section 4), our runtime management and secure loading architecture (Section 5. We then provide an evaluation (Section 6), describe our case study of the Orbot Tor client (Section 7), describe other related work (Section 8), and then conclude (Section 9). We are also providing the code for the entire implementation (compilation, applications, FPGA system, and Android enhancements) in a git repository: `https://github.com/nsr-colorado/cloud-rtr`.

2 Motivation (Why an FPGA)

The premise of incorporating an FPGA into a smart phone lies in the general benefits of an FPGA – that it provides hardware-level programmability which will enable phone manufacturers to defer some decisions about tradeoffs to the end user and enable developers with the ability to innovate in the hardware space.

Here we discuss a few examples that help motivate an FPGA within a smart phone, including a description of a demonstration application that we implemented for each of these categories.

2.1 Architecture enhancements

For our first set of motivating examples, we present several architectural enhancements that have been proposed in the research community that each required a hardware plug-in and were targeted at a server. With our work, similar benefits could be brought to a smart phone. It is important to note that FPGAs are not limited to streaming and highly parallel processing (though they do excel at that). These types of applications can implement security functions, which are supported by our secure loading technology (discussed in Section 5).

CoPilot: CoPilot [57] is a PCI card designed to detect rootkits. As rootkits execute at the highest privilege, detection mechanisms at the same (or lower) privilege are presented with a significant challenge. The CoPilot PCI card is independent of the processor and operating system and has access to all memory via the PCI bus. Rootkit detection (or more generally, security applications) have tremendous potential with the introduction of an FPGA within a smart phone.

Somniloquy: The Somniloquy [27] work observed that the energy consumption on servers was impacted by a number of low-rate type of tasks that prevented the servers from entering the power saving states. As such, they proposed a small, low-power processor that could perform these tasks, and if needed, trigger the main processor to exit a low-power state. In the case of a smart phone with an FPGA, similar types of activity has been observed in smart phones [31], so a small co-processor in the FPGA fabric could provide a solution (while also enabling the main processor to shut off completely). We leave full exploration of power as future work.

As a demonstration of architectural enhancements, we have implemented a memory scanner module, as a simplified proxy for a CoPilot-like function, that scans our device's system address space.

2.2 Software-defined Radio

A great deal of research has resulted in many innovations in wireless communications which allow wireless interfaces to have better performance or more functionality. Research papers in this space commonly use FPGA platforms (such as the WARP Board [28, 53]), or devices to interface to high performance desktop machines (such as the USRP [22]) in order to meet the needs of the new innovation. While these papers provided promising research results, there is little opportunity for deployment – requiring the researchers to commercialize the technology, or get adoption from a major chip vendor.

With a smart phone that has an FPGA along with a more flexible radio front end (*e.g.,* a tunable antenna), developers of a new communication protocol could simply create an app, enhancing the impact of the research. This architecture also has benefits for production systems, as existing devices could be upgraded to new wireless systems without requiring replacement, such as upgrading such a system from 3G to 4G wireless technology.

As a demonstration of an SDR application, we have included an example implementation of a Carrier Phase

Recovery Loop for a single carrier Quadrature Amplitude Modulation (QAM) demodulator. QAM is a representative building block in signal processing applications including many real-world modulation systems.

2.3 Cryptographic and Parallel Processing

FPGAs have the ability to perform large amounts of processing in parallel. This allows them to achieve higher throughputs and lower latencies.

An exemplary application for FPGA acceleration on a smart phone is cryptographic processing, as it both faster in an FPGA and widely used – including the encryption of internet communication using SSL and communication protocols such as Tor [37], the accountable internet protocols [29], and Named-data Networking [44] For example, an FPGA (Altera Stratix V) was shown to be 520 times faster than a general purpose processor (Intel Xeon E5503) for AES encryption (and 15x speedup over an AMD Ratheon HD 7970 GPU) [4]. While the exact numbers will depend on a number of factors, this is illustrative of the potential.

Parallel processing goes beyond cryptography. One recent example used an FPGA based server [10] to implement common functions used in analytics (search, fuzzy search, and term frequency), and in each case demonstrated that it would require 100-200 servers running Spark [65] to match the performance. This example is really geared towards cloud scale applications, but we believe this would allow us to perform some analytic processing locally (on the phone) without needing to send private data to some cloud based backend.

As an example of this type of application, we have implemented a 128-bit AES encryption module that can encrypt an arbitrary number of 128-bit contiguous regions of memory. We also incorporated this AES module into the Orbot Tor client (Section 7).

3 Past Attempts

A central challenge in reaching our vision relates to how to share the FPGA between applications and the system. That is, we wish for multiple apps to be able to simultaneously use some of the FPGA's programmable fabric, while at the same time allowing the operating system to use some of the programmable fabric as well (*e.g.,* to connect to some I/O devices).

The core concept required is run-time reconfiguration, or the ability to dynamically change the FPGA's configuration (completely or partially) at run-time while it is still operating. Despite over a decade of research in run-time reconfiguration [33, 39, 48, 34, 36, 51, 62, 43] there has yet to be a practical solution which would enable hardware modules from various sources to be loaded onto a variety of platforms.

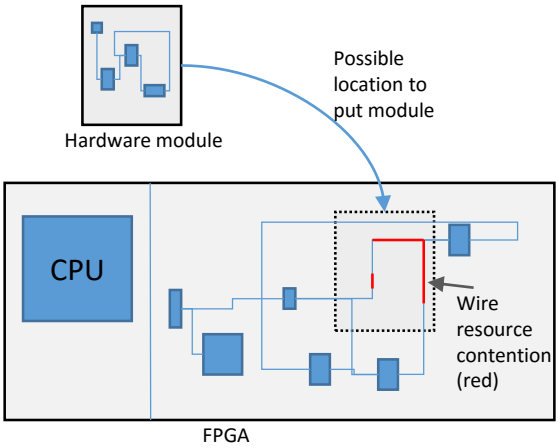


Figure 2: Example of partial reconfiguration in a running FPGA configuration.

3.1 Why is sharing an FPGA difficult?

The main challenge in achieving FPGA sharing is ensuring that the apps' modules in the FPGA do not conflict with each other, or with other logic that is present in the FPGA. FPGAs are difficult to share because a complex mapping of resources must occur in order to generate a configuration for an FPGA. This is because application's logic must be mapped to physical resources in the FPGA, and connections must be made between these locations, just as in any physical circuit.

As an example, consider Figure 2, which illustrates a single module to be loaded into an FPGA at run-time. The dotted area indicates one possible location to put that module. As indicated, however, there will be contention for resources – *i.e.,* this module cannot co-exist with the current FPGA configuration. Because of this, the partial reconfiguration mechanism supported by the vendors (Altera and Xilinx) comes with great restrictions – the modules can only work with a single design (in our case, they wouldn't work across phone architectures), and they can only be loaded into a single location. These restrictions make partial reconfiguration unusable in its current form to enable apps with hardware.

More general run-time reconfiguration approaches have been proposed in the research community that fall into one of two categories, which we describe next. In general neither of these approaches are practical.

3.2 Soln. 1: Run-time Place and Route

The first approach is to perform place and route at run time [60] [46] [56] (rather than when it is normally performed – at design time). As background, place and

route is a computationally expensive task of first mapping logic elements from the design (place) and then determining a collection of wire resources to use to connect the logic elements from the design (route).

This approach enables reconfigurable modules to be created entirely separate from the FPGA configuration. They can be loaded into the FPGA by being placed around existing hardware and connected with free wiring resources.

This is a general approach and supports our model, but there are two major problems. First, place and route can take a long time, depending on both the size of the reconfigurable module as well as the sparseness of the current FPGA configuration – *i.e.,* if there are few resources available, it will be a more difficult task to find a solution. The implication relates to the second problem – that a solution is not always possible, which means that the app would fail to load.

3.3 Soln. 2: Slot-based Reconfiguration

The second method that has been proposed also seeks to support a general approach where the static design and the reconfigurable modules can be created independently. This approach does so by reserving empty and identical areas in the static design [38] [49] [42]. These areas, or slots, are analogous to PCI slots on a motherboard, where any card can be plugged in independent of the processor. In this case, the 'cards' are partial bitstreams (a binary file used to configure an FPGA). Two constraints emerge:

Partial bitstreams need to be relocatable – So that a partial bitstream can be loaded into any slot, each area needs to be identical. This is not difficult from a logic standpoint as FPGAs are fairly regular structures. In order for the static and reconfigurable portions to be able to communicate, however, there need to be wires that cross the boundaries which, in turn, need to be identical for each slot. This puts incredible strain on the creation of the static design, to the point of not being practical (because place and route becomes very constrained if certain circuit elements need to use certain physical wires).

Partial bitstreams cannot conflict with the static design – That is, when loading a partial bistream, it cannot, for example, use a wire, that the main system design used (and vice versa). To achieve this, the static design is highly constrained to reserve areas such that no logic is used (generally, easy to achieve) and such that no wires are used (in Figure 2, this would mean that static portion of the design would not have been allowed the wires that are in the dotted area). Such constraints are ultimately possible (through a painstaking process of reverse engineering and over-constraining), but highly constrains the static portion of the design – forcing wires to be routed around these areas, causing them to be extra long and resulting in congested areas.

In short, this is a good abstraction, but not practical.

4 Cloud RTR: A Practical Approach For Sharing the FPGA

In order to realize the apps with hardware vision, we need two things. First, we need a mechanism to be able to share the FPGA resources – *i.e.,* a practical run-time reconfiguration mechanism that overcomes the limitations of past solutions in terms of usability and deployability. Second, we need a mechanism to be able to manage the apps at run-time. Here, we describe our novel solution for enabling FPGA sharing, and in Section 5 we describe our system support for run-time management of apps.

4.1 High-level Overview

Central to our design, we adopt the general idea of slots – that is, reserved areas within the FPGA where modules can be loaded. As previously mentioned, we believe this is a good abstraction, but the previous realizations of it are not practical. The key difference with our approach is that our slots are less constrained – only logic resources need to be left free (which is easier), but the wiring resources within these areas can be used by the static design logic (*i.e.,* the portion of the FPGA configuration that does not change and provides system functionality for different phones). Other key differences with our approach are that the reconfigurable modules can (i) work with multiple slot sizes, (ii) work with multiple slot signaling interfaces, and (iii) be targeted at various end-systems.

The key idea to enable this is that by leveraging the delivery model of mobile apps (*i.e.,* via an app store), we can effectively merge the modules into various static designs in the cloud, before delivery to the end user. We call this Cloud RTR (RTR for run-time reconfiguration). As illustrated in Figure 3, each phone manufacturer and app developer would submit their design to the Cloud RTR system, and the Cloud RTR system would perform a compilation step to enable a general run-time reconfiguration mechanism.

In this section we describe the architecture of the phone in order to support this model (Section 4.2), how the apps are designed to work within the framework (Section 4.3), and finally discuss how Cloud RTR performs the compilation (Section 4.4).

4.2 Static (Phone) Design Architecture

The key requirement for the phone's design lies in the ability to support interfacing the reconfigurable modules

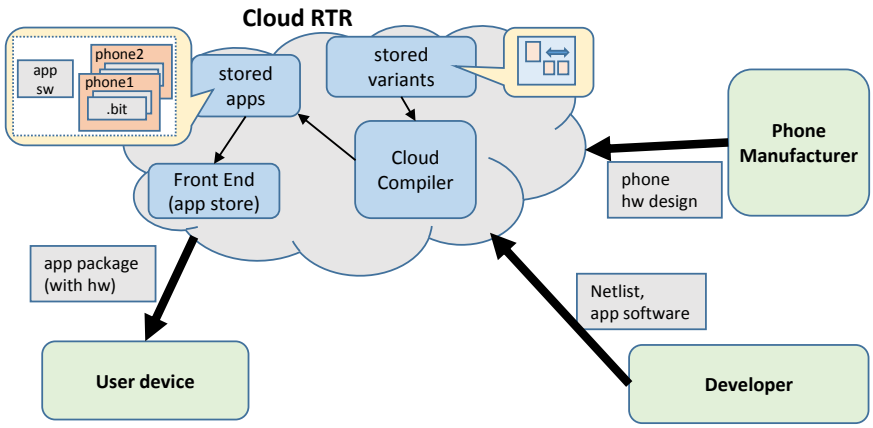


Figure 3: Cloud RTR approach to the generation and deployment of apps with hardware

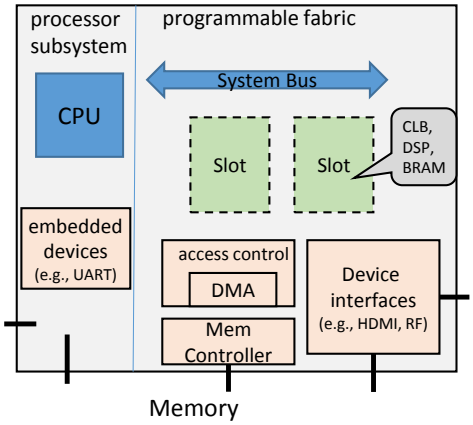


Figure 4: Example static FPGA design.

with the rest of the system resources. Illustrated in Figure 4, and described below, are the main components.

Slots

Slots should have enough of all types of resources to be useful. Today's FPGAs can contain (i) configurable logic blocks (CLBs), which can implement any logic function of N inputs, (ii) block random access memory (BRAM), which are small memory elements (*e.g.,* 36 Kb in the FPGA we use for implementation), and (iii) digital signal processing (DSP) blocks, which are custom building blocks geared toward signal processing applications.

Slots will also need to be able to access various system resources and expose an interface for communication with the processor. For this, we expect that all slots will allow access to (i) a system bus for communication with the processor, and (ii) a direct memory access (DMA) controller for access to system memory.

Module-to-Memory Interface

In order to provide performance benefits, the modules need to be able to directly access CPU-accessible system memory. A DMA controller that is accessible by the hardware modules would allow for modules to access system memory without involving the processor (providing the greatest performance and flexibility). To achieve this, we also need a security module which performs access control – that is, one which limits what memory each hardware module can access and is configured by the operating system.

Processor-to-Module Interface

The ability to stream from memory will be important, but the processor also needs to be able to directly interface to each module. This interfacing is achieved through the use of, for example, a system bus (such as the ARM-based Advanced eXtensible Interface, or AXI).

Device interfacing and other misc. logic

The rest of the static design will include interfacing to the various devices that will connect to the FPGA. Some devices, such as a UART, may have interface logic included in the processor sub-system, but the rest, such as interfacing to a tunable antenna, may go through the programmable fabric with custom logic to interface with it. These devices will be connected to the general interface of the slots, allowing for manufacturers to include custom peripherals without requiring new slot definitions.

4.3 Reconfigurable (App) Module Architecture

In the previous slot-based approaches, the reconfigurable modules are designed for a specific slot design (device, interface, etc.). In our approach, we abstract away the ultimate target such that app developers can develop reconfigurable modules that can be loaded onto a variety of platforms. Of note, the reconfigurable modules in our approach can (i) work with multiple slot sizes, (ii) work with multiple slot signaling interfaces, and (iii) be targeted at various end-systems.

Here we describe the design of an app, with the various components illustrated in Figure 5.

App Hardware

The first major component is the **app hardware**. In order to match the skills of app developers, we focus on the high-level synthesis (HLS) design flow [23] that has emerged in recent years which allows developers to use a high-level language (*e.g.,* C) to describe hardware modules[2]. *What this means is that the argument that FPGAs are hard to design for, and therefore not accessible to the software app developers, is quickly becoming invalid.*

The app hardware (in this example) is written as a C++ function, *example()*. The parameters to the function describe the interfaces to the rest of the system, such as char arrays (*e.g.,* var1), which describe memory mapped registers accessible to the processor or streaming memory interfaces (*e.g.,* var2, which has the type hls::stream), that allow for streaming data from memory (when connected to DMA hardware in the static design). This description is valid C++ code that can be compiled and tested as software which can simplify hardware testing.

While there will need to be some consideration by developers, in general developers will not need to be fully aware of the hardware architecture. For example, the exact bus signals for communicating with the module are not directly used, but are instead inferred based on the types on the function parameters (such as how to perform data transmission handshakes or send valid signals). With this, the same module could actually target various hardware interfaces (*e.g.,* if different handshake protocols or signals are used, or if different bus widths are available). Developers do need to consider the size (resource utilization) of their hardware modules to ensure they will fit in a particular slot size. We envision standard slot sizes will emerge (much like screen sizes), and in our design flow we allow for modules designed for one slot size to always be instantiated in a bigger slot.

App Software

The **app software** that the developer writes will be mostly the same as current apps (*e.g.,* written in Java for Android apps). The only difference is the loading of and interfacing with the hardware module. To load, the app will submit a request to a system service to load the bitstream (*e.g.,* via an intent in Android).

To interface with the module, the app will use the functions in the user-level driver generated by the FPGA vendor's high-level synthesis tool when synthesizing the design (the process which generates the FPGA hardware from the C++ code). This driver is low-level code that runs within the same process as the application and provides functions that can be used to interface with the reconfigurable module. Functionality includes mapping memory regions (*e.g.,* via *mmap()*) that both the reconfigurable module and the app will access. It also provides functions to access the various registers (the char[] variable) through functions like *set_var1()*.

[2]The developer can use a hardware description language, but will then need to manually provide the interfacing hardware and software, which are automatically created with high-level synthesis.

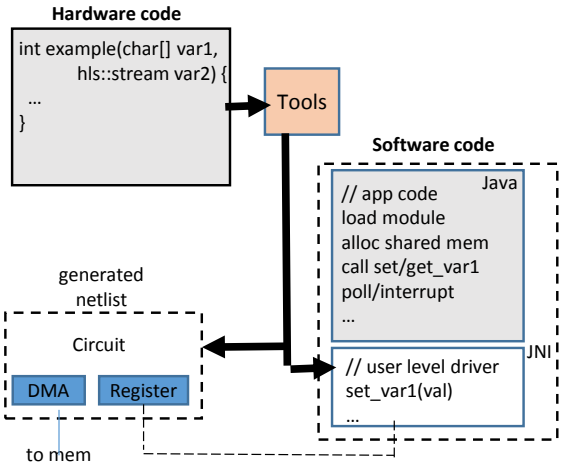


Figure 5: Example app design.

4.4 Cloud Compiler (in the App Store)

The Cloud RTR compiler is responsible for ensuring that an app's hardware module(s) can be loaded into a variety of target devices (smart phones). Rather than working around the limitations of the vendor tools, we work within their constraints, resulting in a practical solution. Recall that the vendor tools have a partial reconfiguration design flow which has the constraints that a module can only be used for a specific static design and target FPGA and for a specific location within that static design. Working within that, the Cloud RTR compiler will simply use the vendor tools to compile the module for every static design variant and for every possible slot within each variant.

The end result is a data structure stored within the app store that looks like the following (where a.bit...e.bit are individual partial bitstreams):

```
[phone 1:
  [slot1:a.bit, slot2:b.bit, slot3:c.bit]]
[phone 2:
  [slot1:d.bit, slot2:e.bit]]
```

When an app is downloaded to a given device, the Cloud RTR system will repackage the application with the set of device-specific bitstreams (possible since the app store has knowledge of a user's device). In Android, for example, apps are packaged in an Android Application Package (APK), which will now include module bitstreams as extra resources for apps that use hardware. To get a rough idea of how this impacts the size of an APK, for the case study we describe in Section 7, the hardware

module bitstream is 904KB, the Orbot APK of the version we modified is 5.5MB (before any added hardware), and the latest Orbot release is 11MB.

We show that this brute-force approach is quite practical in Section 6. As such, it provides a general approach that is deployable and usable today. In addition, we also envision a large amount of reuse of both static designs and hardware modules (*e.g.,* by using precompiled libraries). Just as SoCs are oftentimes reused between different mobile devices, there is no need to have a distinct static design for different devices unless a particular device requires some custom technology.

5 Run-time Management of Apps

With the ability to share the FPGA, as provided by Cloud RTR, we now discuss how our system manages the apps' hardware within the Android operating system. This has two aspects – (i) how to dynamically manage the loading and unloading of hardware modules for various apps, and (ii) how to ensure the modules cannot compromise the running operating system and vice versa. We achieve the dynamic management by modifying the Android operating system to include a system service that manages the loading and unloading of modules and enables application software to access these modules. Our secure loading system takes advantage of some hardware security features of modern FPGAs and some hardware in the static design in order to provide the needed security.

5.1 Dynamic Module Loading Service

To support apps with hardware, there needs to be system support for loading hardware modules into the FPGA. The operating system will have access to our secure loading system (described in the next section) that can take a hardware module compiled using the Cloud RTR system and load it into the FPGA. However, user applications will not have direct access to this system.

User applications will instead submit requests through a privileged *hardware loader* system service. Upon loading and initialization of the app, the service will be provided with the location of the app's hardware module files. The service will then choose an empty slot, select the module compiled for this slot, and use the secure loading module to load the module into the FPGA. In the case where no slots are available, the operating system can create 'virtual' slots by time-slicing existing slots. Given the slot reconfiguration time, we do not expect to swap app hardware as frequently as app software, but we see this is an area for future consideration.

We can implement virtual slots by using the readback capability of FPGAs to store the running configuration of modules, and developers can provide custom unload functionality to aid the readback system in storing difficult to access state (specifically, certain FPGA memory is more difficult to access). Applications that would be disrupted by time slicing can be specifically flagged as unsafe to swap, but the number of these applications running simultaneously should be restricted.

The time to load a hardware module provides an estimate of the time needed to context switch a hardware module. This time is a function of the size of the hardware being written to the FPGA, which we measured to have an average throughput of 37 MiB/s. This leads to a latency of approximately 100 ms for a 4 MB static bitstream, or 27 ms for 1 MB hardware module.

The hardware module is presented as a devfs character device in the Linux /dev directory (when using Android). The hardware loading service will set file permissions to ensure that only the application that requested the loading of the hardware module can access it.

5.2 Secure Loading

We introduce a secure loading mechanism to provide support for protecting both the operating system from app hardware and the FPGA configuration from the operating system (*e.g.,* a rootkit). Our secure loading mechanism is an extension of secure boot technology where (i) we disable the processor's connection to the configuration ports of the FPGA, and (ii) we add a module within the static design to support loading of app hardware. For space purposes we can only sketch the high-level overview of the secure loading.

Threat model and assumptions: We assume that any code running on the CPU, including the operating system, can be malicious. We also assume that even with the support of modules such as the trusted platform module (TPM) to protect the booting of the operating system, malicious code can be executed at run-time that can compromise the operating system. We assume that some reconfigurable modules will be untrusted and potentially malicious. We assume that some reconfigurable modules will be trusted (from trusted sources, such as the phone manufacturer) and will not be malicious. Finally, we assume that the static FPGA configuration is trusted and correct at boot time (through secure boot mechanisms supported by modern FPGAs [64]). With this, we assume the keys of the app store and an additional trusted party are present at boot and cannot be modified.

The secure loading can be summarized as follows:

- Secure boot technology of the FPGA will load a trusted FPGA configuration as well as ensure the operating system is known to be good at boot.
- As part of the secure boot, the processor will have its access to the configuration ports disabled (*e.g.,*

the processor configuration access port, PCAP, and the internal configuration access port, ICAP).

- A hardware secure loading module, part of the static configuration, will accept requests to load a reconfigurable module from the operating system .
- Every module will be signed by the app store. From this the hardware secure loading module will verify the signature in order to ensure that the configuration refers to is the slot to be loaded (preventing the case where an app was modified to include a configuration which overwrites parts of the FPGA other than the allocated area).
- For trusted modules (which will signed by the app store and by an additional trusted party), the hardware secure loading module will verify the signature and if it succeeds, configure the memory access control to allow the module to have access to system memory. Trusted modules will not be able to be unloaded or overwritten without a reboot.
- For untrusted modules (which will be signed by the app store, but not by an additional trusted party), the hardware secure loading module will configure the memory access control to restrict access to memory.

With this process, we get the following properties:

- Modules will be correctly loaded only into the slots for which they are supposed to be loaded and not overwrite any static configuration.
- Code in the operating system cannot load a module in a slot which gets access to system memory (preventing app hardware from helping apps bypass system protections, or otherwise harm the system).
- Code in the operating system cannot overwrite or unload a trusted module (preventing rootkits, for example, from unloading security modules meant to detect the presence of rootkits).

6 Evaluation

There are two main questions to answer, which we discuss in this section:

Is there value in apps with hardware?

There's general acceptance that hardware will be faster than software[3]. The question we seek to answer here is whether the same performance benefits are retained when we consider it within a system (*e.g.,* does crossing the hw-sw boundary make things worse).

[3]That's not really the focus of our paper – we take the stance that there are places where each wins (FPGA, CPU, GPU) and that heterogeneous architectures are good, and more importantly open programmability is what drives innovation.

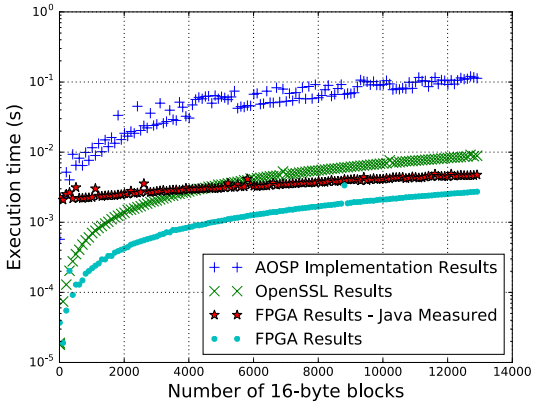


Figure 6: The execution time to perform an AES encryption for a range of data sizes – from 10 to 13000 contiguous 128-bit (16-byte) segments of memory.

Is the cloud compilation of Cloud RTR practical?

As mentioned, rather than continuing the path of runtime reconfiguration research, which leads to creative, but impractical solutions, we aimed for a solution which was highly practical and deployable today. This resulted in a brute force approach. Here, we ask whether this is itself practical by examining the processing required to support the app market ecosystem.

6.1 Application Performance Acceleration

Performance acceleration is one of the benefits of using an FPGA. Of course, we also believe that new applications are now enabled, such as our hardware-based memory scanner. We focus on performance here as a concrete demonstration with a quantitative evaluation.

We focus on three key application domains that are enabled. Each of these applications consists of a hardware module written in C++ using high-level synthesis and an Android application that interfaces with the module. We run the hardware module through the Cloud RTR compilation platform targeting the static design for our development board. The end result is an APK that can be loaded into our demonstration board. Each application fit within our slots, which we defined at 12% of the overall FPGA area – a number resulting from dividing the remaining area after what is needed for the static design by six available slots.

For our experiments, we protoyped a mobile device using the Zedboard development board, which integrates a Xilinx Zynq 7020 FPGA [25] that has an embedded dual-core ARM Cortex-A9 CPU. We based our Android services on the Android 2.3 and 5.0.2 operating systems that were already ported to our device.

6.1.1 Cryptography: AES

In Figure 6, we compare three different AES implementations using our development board, and running An-

droid as the operating system on the CPU[4]:

- **FPGA (ours)** – an AES FPGA reconfigurable module accessed by an Android application in native code using the the Java native interface (JNI).
- **OpenSSL** – the OpenSSL AES implementation which we interfaced directly from a C application.
- **Android AES** – the AES implementation provided to Android applications by the AOSP.

This figure shows the execution times of the AES implementation (which we derived from [1], though we also experimented with versions from Apple [2] and NIST [3], which had identical performance) for a range of data sizes to be encrypted, varying in size from 10 to 13000 contiguous 128-bit segments of memory. It can be seen that the FPGA implementation is on average three times faster than the OpenSSL implementation, and is approximately 12 times faster than the AOSP. However, the execution time of the FPGA module as measured by Java (marked in red circles) and executed using the JNI is longer than the execution time of the same module when executed directly by a C program (marked in blue diamonds). This is likely due to overhead entailed in copying memory to the JNI function call and transferring control to the JNI. This can potentially be alleviated using Java direct byte buffers passed directly to the JNI function, but is deferred to future work.

6.1.2 Software-defined Radio: QAM

This application can process a signal stored in a contiguous memory region and produce an output signal that is stored into another contiguous region. In a live smart phone, the static design would place the signal off of the antenna into buffers in memory, notify the Android application of a buffer being full, and the application would pass this data to the QAM module. The number of samples the QAM block can process determines the sample rate of the radio application. We implemented this module by modifying (to be compatible with our Cloud RTR system) a reference Xilinx project [55], which comes with C++ code that can be executed in software or run through high-level synthesis to produce hardware.

As shown by Figure 7, the hardware implementation is several orders of magnitude faster than the software implementation. The hardware implementation achieves an average throughput of approximately 5 Msps (mega-samples per second), while the software implementation only achieves and average of approximately 500 samples/s. The Xilinx application notes claim a throughput value of 50 Msps [55], which is likely achievable due to the fact that the hardware device is intended to process

[4] We also performed the OpenSSL benchmark in Ubuntu Linux to confirm that the Android OS does not institute a performance penalty.

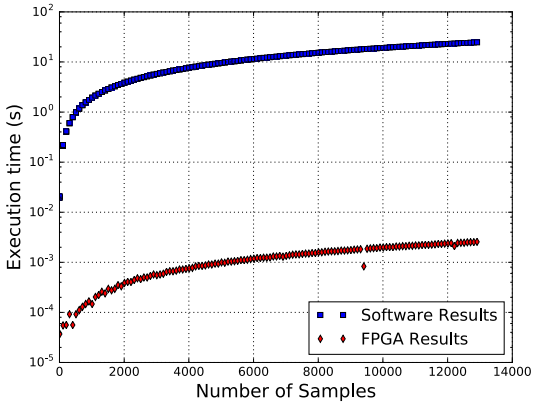


Figure 7: The execution time to process a different number of samples with our QAM application.

data received directly from an analog-to-digital converter (ADC), whereas our implementation has been retrofitted to stream data from system memory.

6.1.3 Memory Scanner

Our final application is a simple implementation of a hardware memory scanner that searches our device's address space for occurrences of a 16 byte string. We wrote custom C++ code that we ran through high-level synthesis for this implementation.

Using the LMbench testbench [50], we instituted a memory benchmark that measured the throughput of our device's memory while under normal operation and while the hardware memory scanner was executing (which is constantly reading from memory). Using this benchmark, we measured a 2.7% reduction in performance for read operations and a 5.5% reduction in performance for write operations.

6.2 Cloud Compilation Resources Needed

We propose performing compilation of the reconfigurable module in the cloud as part of the process to upload to the app store. To understand the feasibility of this, here we evaluate the amount of computing resources needed to sustain an 'apps with hardware' ecosystem.

The metric of interest is how long it takes to compile a single reconfigurable module for a given static design. Recall that a static design is the base design that roughly corresponds to a system on chip used for a given smart phone. These static designs have open areas (slots) for placing reconfigurable modules.

For all experiments, we used a server with an Intel Xenon CPU (2.1 GHz, 6 cores, 48 GB RAM).

6.2.1 Single App for a Single Static Design

An app with hardware uploaded to the app store must have its hardware modules compiled for each slot that it

# of slots	Compilation Time (min)	Throughput (apps/day)
2	11.92	121
3	14.93	96
4	19.02	76
5	24.21	59
6	28.23	51

Table 1: Compilation time and number of apps a single server could service per day.

6 Slots Requirement	% of Jan 2016 Apps that Use Hardware		
	0.1	1	10
	# of Apps Uploaded per Day		
	5	52	520
# of Static Designs	# of Machines Required to Compile RMs		
1	1	1	10
10	1	10	102
100	10	102	1020
1000	102	1020	10200

Table 2: Number of servers required to support the compilation requirements of Cloud RTR, assuming designs with six slots.

can be placed into. However, certain steps in the process do not need to be performed for each slot – *e.g.,* synthesis needs to be performed once for each module, then for each slot, the synthesized module needs to be placed and routed. For this evaluation we used an FFT module, which is a highly regular structure and enabled us to adjust its parameters to effectively alter its size to fill up any slot size we experimented with.

Table 1 shows the total time to compile a single application's reconfigurable module for static designs with two to six slots (each slot is defined as 12% of the overall area of our FPGA, as previously mentioned), as well as the extrapolated throughput (the number of apps that could be compiled per day on one server given this compilation time). We chose up to six slots (in contrast to the 60-100 slots in [38]) as we believe each to be big enough to implement a reasonable module within a single slot.

6.2.2 Compiling All Apps

Using the calculated throughput, we can now estimate the amount of computing resources needed to service the entire app ecosystem.

First, we need to know how many apps are uploaded each month. The company AppFigures provided us with the Google Play Store application upload figures for the entire year of 2014, with a total of 1.43 million apps at the end of 2014, and an average monthly growth of 6.10%. While we are unsure if this growth rate persisted over the past year, we use it to estimate the current needs. Using this average monthly app growth, for the month of January 2016, 177,521 applications are predicted to be uploaded into the Google Play Store.

Table 2 shows the number of machines required to service monthly demands for compiling apps with hardware, for six slots as an example (2-5 slots would be proportionally less). Each table varies the number of apps with hardware uploaded each day based on the percentage of applications that require hardware (0.1%, 1%, and 10%), as well as the number of static hardware variants – for the sake of illustration, we assume from one to 1000 variants, with each interval increased by a factor of ten (we expect the number of variants to be on the low end, as static bitstreams can be reused between devices, just as phones today use a small set of SoCs, and not every device will require a new static bitstream).

The cloud provider will easily be able to support the lower end of the spectrum internally. On the upper end, the cloud provider might look to relieve the computation burden by offloading to the phone manufacturers to compile for their own variants.

7 Case Study: Orbot Tor Client

Developing our demonstration applications from scratch allows us with to design it to use an FPGA natively. Here we explore modifying an existing, complex application to make use of a hardware module to understand the inefficiencies that may result.

We chose to modify the Orbot [17] Tor [37] client for Android. Tor is an anonymization network that allows for a user to access the internet without disclosing their source IP address, making identifying and tracking their internet traffic very difficult for third parties. A Tor client creates a circuit through this network and encrypts their traffic separately for each node along the path to prevent eavesdropping during transmission. Because of this extensive use of encryption and based on notes by the Orbot developers mentioning that AES is one of the areas to optimize Orbot [21], we see this as an ideal case study.

Our AES module implements the CTR (counter) mode of operation on top of a standard AES block cipher that is an equivalent to the OpenSSL CTR implementation used by Tor. In order to integrate our AES accelerator with Tor, we replaced all calls to OpenSSL AES encryption with calls to the FPGA accelerator, which proved to be a fairly minor modification. We also needed to ensure that all data that was to be encrypted was located in a contiguous memory region with a known physical address, which required us to replace all *malloc()* calls with calls to a custom memory allocator, and leverage a memory region that we reserved from the kernel.

With this, the application is able to make use of the FPGA resources and operate correctly. However, there are inefficiencies remaining due to (i) the overhead required to allocate memory in the reserved region, (ii) the overhead in accessing this memory, as it is implemented using memory-mapped I/O, and (iii) the fact that certain

memory system calls (*e.g., malloc()*, *memcpy()* and *memset()*) are incompatible with the current memory mapped implementation – which would require more extensive modifications to the code to resolve. Even so, this provides us with great insight into how apps should be designed to capitalize on the FPGA resources and is an area for future improvements.

8 Related Work

Although there are no existing systems that implement all of the functionality of our Cloud RTR system in mobile devices, there has been much work done in reconfigurable computing in other contexts, including several different attempts with Android.

Of note from previous reconfigurable computing research is the BORPH system [62], which attempts to create operating system extensions in Linux for FPGA operations, and uses Berkeley's BEE2 system [35], and the more recent Connectal framework [47], which can automatically generate HW/SW interfaces during hardware development. These systems, however, do not address application distribution or FPGA resource sharing.

In terms of mobile systems research, some proposals have been made, such as the rSmart system [63] and the work from Smit et. al. [61]. Smit et. al. proposes a similar hardware architecture to the Zynq-7000 architecture, but does not present an operating system integration or a deployment system. The rSmart system only presents a high-level sketch of a system similar to ours, but no details on implementation or integration are provided. Our system builds upon this research to create a general system that is deployable using existing technology.

There has also been recent advantage in reconfigurable cloud platforms. For example, Microsoft's Project Catapult makes use of of FPGA peripherals in data centers to accelerate web searches [58] and neural networks [54], and Intel's acquisition of Altera [13] is leading to x86 CPU architectures coupled with FPGAs [40]. Microsoft's solutions, however, are only single-application hardware accelerators, whereas our system allows for usage in general applications. Intel's system is more general, but has not been released publicly, but does claim to use OpenCL [16] as the software interface.

Our work is complimentary. For example, OpenCL can be used on mobile devices with support from major hardware manufacturers, such as ARM, Intel and Qualcomm, [14] [20] [8] [12] [18], and can even be used with our system by using a compatible hardware module. OpenCL's main limitation is its focus on parallel acceleration, which does not enable new architectural enhancements, such as our SDR or security applications.

Reconfigurable Android devices and systems have also been proposed, such as Google's Project Ara [11], among others, including various other modular phones [19] [9] [15]. These modular phone systems allow for reconfiguration and upgrading of smart phone *physical* components, similar to how personal computer components can be upgraded. However, these modular architectures can only be reconfigured manually by the user replacing the physical modules, whereas our system allows for dynamic and custom reconfiguration by software.

Finally, the Android OS has been ported to the Zynq-7000 in several projects, such as the work of Barbareschi, et. al., among others [32] [7] [6] [24]. However, with the exception of the work of Barbareschi, et. al., these projects only port the OS to a new device. The work of Barbareschi, et. al. only extends this work to create an Android-compatible custom accelerator to address a single problem, whereas our system allows for any general software to create their own custom hardware modules.

9 Conclusions and Future Work

In this paper we presented the concept of 'apps with hardware' where, with the introduction of an FPGA into a smart phone we can enable app developers to innovate in the architectural (hardware) space, as they can today in the software space. Our Cloud RTR cloud based compilation mechanism overcomes past limitations of using the FPGA in a general way and does so without requiring any modifications from the vendors (making it deployable today). Our Android-based run-time application management system enables the dynamic management of the execution of apps (and their use of the available hardware), and provides a secure loading mechanism.

There is a great deal of possible future work. First, performing a thorough power analysis in a fair manner will provide great insight into both the benefits and needed system support, and building more apps to capitalize on the new capabilities would likewise provide great insight. Second, we wish to work with additional tools from other vendors (*e.g.,* Altera) and operating system platforms to explore implementation differences and with a partner to develop an actual prototype smart phone system (rather than the Zedboard) to further understand its viability. Finally, we wish to further investigate memory management techniques for better optimization.

10 Acknowledgments

This research was supported in part by NSF SaTC grant number 1406192. We would also like to thank Phil James-Roxby and Derek Woods for their guidance, and Xilinx for their hardware and software donations.

References

[1] http://programmablelogicinpractice.com/?p=87.

[2] http://www.opensource.apple.com/source/CommonCrypto/CommonCrypto-55010/Source/libtomcrypt/src/ciphers/ltc_aes/aes.c.

[3] http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[4] 40Gbit AES Encryption Using OpenCL and FPGAs. http://www.nallatech.com/40gbit-aes-encryption-using-opencl-and-fpgas.

[5] Altera socs. https://www.altera.com/products/soc/overview.html.

[6] Android 4.2.2 on zynq getting started guide. http://www.wiki.xilinx.com/Android+4.2.2+On+Zynq+Getting+Started+Guide.

[7] Android on zynq getting started guide. http://www.wiki.xilinx.com/Android+On+Zynq+Getting+Started+Guide.

[8] ARM Mali OpenCL SDK. http://malideveloper.arm.com/resources/sdks/mali-opencl-sdk/.

[9] Fairphone. https://www.fairphone.com/.

[10] FPGA System Smokes Spark on Streaming Analytics. www.datanami.com/2015/03/10/fpga-system-smokes-spark-on-streaming-analytics/.

[11] Google Project Ara. http://www.projectara.com/.

[12] GPGPU OpenCL API. http://www.vivantecorp.com/index.php/en/technology/gpgpu.html.

[13] Intel Altera Acquisition. https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/.

[14] Intel OpenCL SDK. https://software.intel.com/en-us/intel-opencl.

[15] LG G5. http://www.lg.com/us/mobile-phones/g5.

[16] OpenCL. https://www.khronos.org/opencl/.

[17] Orbot. https://guardianproject.info/apps/orbot.

[18] PowerVR SDK. https://community.imgtec.com/developers/powervr/.

[19] Puzzlephone. http://www.puzzlephone.com/.

[20] Qualcomm Adreno GPU SDK. https://developer.qualcomm.com/software/adreno-gpu-sdk/tools.

[21] Tor source code hacking documentation. https://gitweb.torproject.org/tor.git/tree/doc/HACKING.

[22] Universal Software Radio Peripheral (USRP) by Ettus Research. http://www.ettus.com/.

[23] Vivado high-level synthesis. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/.

[24] Zedroid - android (5.0 and later) on zedboard. http://www.slideshare.net/noritsuna/zedroid-android-50-and-later-on-zedboard.

[25] Zynq-7000 all programmable soc. http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.

[26] Zynq UltraScale+ MPSoC. http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[27] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Proc. USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[28] K. Amiri, Y. Sun, P. Murphy, C. Hunter, J. R. Cavallaro, and A. Sabharwal. Warp, a unified wireless network testbed for education and research. In *Proceedings of IEEE MSE*, 2007.

[29] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *Proc. ACM SIGCOMM*, 2008.

[30] P. Bahl, R. Chandra, T. Moscibroda, R. Murty, and M. Welsh. White space networking with Wi-Fi like connectivity. In *Proc. SIGCOMM*, Aug. 2009.

[31] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: A measurement study and implications for network applications. In *Proc. ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*, 2009.

[32] M. Barbareschi, A. Mazzeo, and A. Vespoli. Network traffic analysis using android on a hybrid computing architecture. In *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing - Volume 8286*, ICA3PP 2013, pages 141–148, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[33] G. Brebner. Circlets: Circuits as applets. In *Proc. IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 1998.

[34] G. J. Brebner. A virtual hardware operating system for the xilinx xc6200. In *Proc. International Workshop on Field-Programmable Logic (FPL)*, 1996.

[35] C. Chang, J. Wawrzynek, and R. W. Brodersen. BEE2: A High-End Reconfigurable Computing System. *IEEE Des. Test*, 22(2), Mar. 2005.

[36] O. Diessel and G. Wigley. Opportunities for operating systems research in reconfigurable computing. Technical Report ACRC99018, Advanced Computing Research Centre, School of Computer and Information Science, University of South Australia, 1999.

[37] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *Proc. USENIX Security Symposium*, 2004.

[38] D.Koch, C. Beckhoff, and J. Teich. Recobus-builder a novel tool and technique to build statically and dynamically reconfigurable systems for fpgas. In *Proc. Field Programmable Logic and Applications (FPL)*, 2008.

[39] S. A. Guccione and D. Levi. XBI: A java-based interface to FPGA hardware. In *Configurable Computing: Technology and Applications, Proc. SPIE 3526*, pages 97–102, Nov. 1998.

[40] P. K. Gupta. Xeon+fpga platform for the data center. *The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL)*, June 2015.

[41] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 195–206, New York, NY, USA, 2010. ACM.

[42] E. Horta, J. Lockwood, and D. Parlour. Dynamic hardware plug-ins in an fpga with partial run-time reconfiguration. In *Proceedings of the 39th conference on Design automation*, June 2002.

[43] E. L. Horta, J. W. Lockwood, and S. Louis. PARBIT : A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Dept. Comput. Sci., Washington Univ., Saint Louis, MO, 2001.

[44] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proc. Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[45] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen. Raising the bar for using gpus in software packet processing. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 409–423, Oakland, CA, May 2015. USENIX Association.

[46] E. Keller. Jroute: A run-time routing api for fpga hardware. In *IPDPS Workshops, ser. Lecture Notes in Computer Science*, volume 1800, 2000.

[47] M. King, J. Hicks, and J. Ankcorn. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 13–22, New York, NY, USA, 2015. ACM.

[48] E. Lechner and S. A. Guccione. The java environment for reconfigurable computing. In *Proc. International Workshop on Field-Programmable Logic and Applications*, Sept. 1997.

[49] M. Majer, J. Teich, A. Ahmadinia, and C. Bobda. The erlangen slot machine: A dynamically reconfigurable fpga-based computer. In *VLSI Signal Processing Systems*, 2007.

[50] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[51] J.-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip. In *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 2003.

[52] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. Netfpga: Reusable router architecture for experimental research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO)*, 2008.

[53] S. Neuendorffer and C. Epifanio. Generic partially reconfigured processor systems applied to software defined radio. In *Proc. of the Software Defined Radio Forum (SDR)*, 2007.

[54] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[55] A. Paek and D. Mackay. Implementing carrier phase recovery loop using vivado hls. `http://www.xilinx.com/support/documentation/application_notes/XAPP1173-carrier-loop.pdf`.

[56] C. Patterson, P. Athanas, M. Shelburne, J. Bowen, J. Suris, T. Dunham, and J. Rice. Slotless module-based reconfiguration of embedded fpgas. In *ACM Trans. Embedd. Comput. Syst*, October 2006.

[57] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. USENIX Security Symposium*, 2004.

[58] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[59] T. Rinta-aho, M. Karlstedt, and M. P. Desai. The click2netfpga toolchain. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 77–88, Boston, MA, 2012. USENIX.

[60] S.Guccione, D. Levi, and P. Sundararajan. Jbits: Java-based interface for reconfigurable computing. In *Proc. Conf. on Military and Aerospace Application of Programmable Devices and Technology*, 1999.

[61] G. J. M. Smit, P. J. M. Havinga, L. T. Smit, P. M. Heysters, and M. A. J. Rosien. Dynamic reconfiguration in mobile systems. In *Proc. International Conference on Field-Programmable Logic and Applications (FPL)*, 2002.

[62] H. K.-H. So and R. Brodersen. A Unified Hardware/Software Runtime Environment for FPGA-based Reconfigurable Computers Using BORPH. *ACM Trans. Embed. Comput. Syst.*, 7(2), Jan. 2008.

[63] N. Soundararajan. rSmart: The Reconfigurable (Real) Smartphone. *Provocative Ideas session of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2013.

[64] S. Trimberger and J. Moore. Fpga security: Motivations, features, and applications. *Proceedings of the IEEE*, 102(8):1248–1265, Aug 2014.

[65] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, 2012.

Testing Error Handling Code in Device Drivers Using Characteristic Fault Injection

Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu
Department of Computer Science and Technology, Tsinghua University

Abstract

Device drivers may encounter errors when communicating with OS kernel and hardware. However, error handling code often gets insufficient attention in driver development and testing, because these errors rarely occur in real execution. For this reason, many bugs are hidden in error handling code. Previous approaches for testing error handling code often neglect the characteristics of device drivers, so their efficiency and accuracy are limited. In this paper, we first study the source code of Linux drivers to find useful characteristics of error handling code. Then we use these characteristics in fault injection testing, and propose a novel approach named EH-Test, which can efficiently test error handling code in drivers. To improve the representativeness of injected faults, we design a pattern-based extraction strategy to automatically and accurately extract target functions which can actually fail and trigger error handling code. During execution, we use a monitor to record runtime information and pair checkers to check resource usages. We have evaluated EH-Test on 15 real Linux device drivers and found 50 new bugs in Linux 3.17.2. The code coverage is also effectively increased. Comparison experiments to previous related approaches also show the effectiveness of EH-Test.

1. Introduction

As important components of the operating system, device drivers control hardware and provide fundamental supports for high-level programs. During driver execution, different kinds of occasional errors may occur, such as kernel exceptions and hardware malfunctions [31]. Therefore, device drivers need error handling code to assure reliability. But in some drivers, error handling code is incorrect or even missed. In these drivers, serious problems like system crashes and hangs may occur when occasional errors are triggered. According to our study on Linux driver patches, more than 40% of accepted patches add or update corresponding error handling code. It shows that error handling code in device drivers is not reliable enough, so testing error handling code and detecting bugs inside are very necessary.

A challenge of testing error handling code is that occasional errors are infrequent to happen in real execution [34]. For example, "bad address" (EFAULT) is a common error should be handled, but it happens only when the memory or I/O address is invalid. Another example is hardware error, which happens only when the hardware malfunctions. Triggering these errors in real environment is very hard and uncontrollable.

To simulate software and hardware errors at runtime, *software fault injection* (SFI) is often used in driver testing. This technique mutates the code to inject specific errors into the program, and enforces error handling code to be executed at runtime. Linux Fault Injection Capabilities Infrastructure (LFICI) [43] is a well-known project integrated in Linux kernel. It can simulate common errors, such as memory-allocation failures and bad data requests. Inspired by LFICI, other fault injection approaches [7, 13, 23, 32] have been proposed in recent years, and they have shown promising results in driver testing and bug detection. However, these approaches still have some limitations in practical use.

- The representativeness of injected faults is often neglected, and most injected faults are random or manually selected. Random faults can not reflect real errors well. Manually selected faults often omit representative injected faults.
- Numerous redundant test cases are generated. In fact, many generated test cases may cover the same error handling code, but they all need to be actually tested at runtime. For this reason, they often spend much time in runtime testing.
- Only several kinds of faults can be injected, such as memory-allocation failures. But these faults can not cover most error handling code in drivers.
- Much manual effort is needed. The kinds and places of injected faults are often manually decided.

In fact, previous fault injection approaches aim to support general software, but they neglect the characteristics of target programs. To relieve their limitations, we should consider the key driver characteristics in SFI. For example, because drivers are often written in C, so built-in error handling mechanisms (such as "*try-catch*") are not supported. For this reason, the developers often use an *if* check to decide whether the error handling code should be triggered in device drivers. This characteristic can help to decide which functions can actually fail and should be fault-injected.

In this paper, we first study Linux driver code, and find three useful characteristics in error handling code: *function return value trigger*, *few branches* and *check decision*. Then based on these characteristics and SFI, we propose a practical approach named EH-Test[1] to efficiently test error handling code and detect bugs inside. Firstly, EH-Test uses a *pattern-based extraction strategy* to extract target functions which can fail from the captured runtime traces of normal execution. This strategy can automatically and accurately extract real target functions to improve the representativeness of inject faults. Then, we generate test cases by corrupting the return values of target functions. Next, we run each test case on the real hardware, and use a monitor to record runtime information and pair checkers to check resource-usage violations. These pair checkers contain the basic information of resource-acquiring and resource-releasing functions, which can be obtained from specification mining techniques [18, 20, 37, 38] and user configuration. During driver execution, system crashes and hangs can be easily identified through kernel crash logs or user observation. After driver execution, EH-Test can report resource-release omissions. We have implemented EH-Test using LLVM, and evaluated it on 15 Linux drivers of three classes. The results show that EH-Test can accurately find real bugs in error handling code and improve code coverage in runtime testing. Comparison experiments to previous approaches also show its effectiveness.

Compared to previous SFI approaches for testing drivers, our approach have four advantages:

1) Representative injected faults. We design a pattern-based extraction strategy to automatically and accurately extract real target functions as representative injected faults. It uses code patterns to decide whether a function can actually fail in driver execution. This strategy can largely improve the effectiveness of SFI.

2) Efficient test cases. According to our study, many drivers have few branches in error handling code, so injecting a single fault in each test case is enough to cover most error handling code. Moreover, our pattern-based extraction strategy can filter many unrepresentative injected faults. Therefore, the test cases generated by EH-Test are efficient, and the time usage of runtime testing can be largely shortened.

3) Accurate bug detection. By injecting representative faults, EH-Test can realistically simulate different kinds of occasional errors to cover error handling code. Moreover, EH-Test runs on the real hardware and uses exact execution information to perform analysis. These points assure the accuracy of bug detection.

[1] EH-Test program can be downloaded from the link: *http://oslab.cs.tsinghua.edu.cn/EHTest/index.html*

4) High automation and scalability. Most working procedure of EH-Test is automated, including target-function extraction, fault injection and test-case execution. And it can support many kinds of existing drivers.

In this paper, we make the following contributions:

- We study the source code of Linux device drivers, and find three useful characteristics in error handling code.
- Based on the patterns of error handling code in drivers, we design a pattern-based extraction strategy to automatically and accurately real target functions as representative injected faults.
- Based on the characteristics, we design a practical approach named EH-Test, which can efficiently test error handling code in device drivers and detect bugs inside.
- We evaluate EH-Test on 15 device drivers in Linux 3.1.1 and 3.17.2, and respectively find 32 and 50 bugs. All the detected bugs in 3.17.2 have been confirmed by developers. The code coverage is also effectively increased in runtime testing. We also perform comparison experiments to previous approaches, and find that EH-Test can detect the bugs which are missed by them. The experimental results show that EH-Test can efficiently perform driver testing and accurately find real bugs.

The rest of this paper is organized as follows. Section 2 introduces the motivation. Section 3 presents the three characteristics of device drivers found by our study on Linux driver code. Section 4 presents our pattern-based extraction strategy. Section 5 introduces EH-Test in detail. Section 6 shows our evaluation on 15 Linux device drivers and comparison experiments to previous approaches. Section 7 introduces the related work and Section 8 concludes this paper.

2. Motivation

To ensure the reliability of device drivers, error handling code should be correctly implemented to handle different kinds of occasional errors. But in fact, error handling code is incorrect or even missed in some drivers, so hard-to-find bugs may occur during execution. In this section, we first reveal this problem using a concrete example and our study on Linux driver patches, and then we sketch the software fault injection technique used in this paper.

2.1 Motivating Example

We first motivate our work using a real Linux driver *bnx2*. This driver manages Broadcom NetXtreme II Ethernet Controller. Figure 1 shows a part of its source code in Linux 3.1.1. The function *bnx2_init_board* calls *pci_request_regions* (line 7906) to request PCI I/O and memory resources when initializing the hardware. If an

occasional error occurs when mapping bus memory into CPU, the function *ioremap_nocache* (line 7937) will fail and return NULL. In this situation, the driver calls *pci_release_regions* (line 8248) to release allocated resources in error handling code. Reviewing the code, the function *kzalloc* (line 7885) is called to allocate kernel-space memory. But this memory is not freed in error handling code, so a memory leak occurs. In fact, this bug still remains in Linux 3.17.2.

In this example, we have three findings. Firstly, error handling code in drivers is often used to release allocated resources and undo recent operations [30]. It is because that many drivers are based on the *fail-stop* model [33], namely a simple error can force the driver to exit. Due to this feature, many bugs in error handling are related to resource-usage violations, such as resource leaks and deadlocks. Secondly, error handling code is often written in a separate segment in drivers (line 8274-8253 in Figure 1 is an example), and different "*goto*" target labels handle different errors. This *goto-based* strategy is recommended by the Linux kernel documentation [44], because this strategy can simplify error handling logic and reduce repeated code. Thirdly, bugs in error handling code are hard-to-find. It is because that error handling code is rarely executed, and maintainers pay insufficient attention to it. In the example, from Linux 3.1.1 (released in November 2011) to 3.17.2 (released in October 2014), the memory leak in Figure 1 had not been fixed. Thus, it is very necessary to reveal and detect bugs in error handling code.

Path: linux-3.1.1/drivers/net/bnx2.c

```
7869. static int __devinit bnx2_init_board(....)
7870. {
          ......
7885.     bp->temp_stats_blk = kzalloc(...);
          ......
7906.     rc = pci_request_regions(pdev, ...);
          ......
7937.     bp->regview = ioremap_nocache(...);
7938.     if (!bp->regview) {
7939.         dev_err("Cannot map register space, aborting\n");
7940.         rx = -ENOMEM;
7940.         goto err_out_release;
7941.     }
          ......
8247. err_out_release:
8248.     pci_release_regions(pdev);
8249. err_out_disable:
8250.     pci_disable_device(pdev);
8251.     pci_set_drvdata(pdev, NULL);
8252. err_out:
8253.     return rc;
8254. }
```

Figure 1: Part of the bnx2 driver code in Linux 3.1.1.

Driver Class	Accepted Patches	Error Handling
I2C	29	13(44.83%)
PCI	38	13(34.21%)
PowePC	42	11(26.19%)
RTC	24	8(33.33%)
Network	598	253(42.31%)
Total	731	298(40.77%)

Table 1: Study result of Linux driver patches.

2.2 Study on Linux Patches

To clearly illustrate the reliability of current error handling code in device drivers, we make a study on Linux driver patches. We manually read patches in the Patchwork project[2] and select accepted patches from them in July 2015. These patches are from 5 driver classes, namely I2C bus drivers, PCI bus drivers, PowerPC drivers, real-time clock (RTC) drivers and network drivers. Among them, we identify those which add or update corresponding error handling code. The result is listed in Table 1. The first column presents the driver class name; the second column shows the number of accepted patches; the third column shows the number and percentage of accepted patches add or update corresponding error handling code.

From Table 1, we find that 40% accepted patches add or update corresponding error handling code. In these accepted patches, many are used to fix common bugs, such as memory leaks and null pointer dereferences. One reason for this phenomenon is that complex control flows and different kinds of occasional errors make it difficult to implement correct error handling code. Another reason is that error handling code is often triggered by specific and infrequent conditions (such as insufficient memory and hardware errors), so developers hardly test it well at runtime.

In brief, current error handling code in device drivers is not reliable enough as we expected, and many bugs are hidden in it. Once these bugs are triggered, serious system problems may occur, such as crashes and resource leaks. Therefore, it is important and necessary to test error handling code in device drivers and detect bugs inside.

2.3 Software Fault Injection

Software fault injection (SFI) is a widely used technique of testing error handling code. It intentionally introduces faults or occasional errors into the program, and then tests whether the program can correctly handle the injected faults or errors at runtime. In this paper, we use SFI to test drivers and detect bugs. To help better understand this paper, we explain several terms about SFI.

[2] Patchwork project. *http://patchwork.ozlabs.org/*

Fault Injection. We inject faults or errors to make error handling code executed at runtime. In this paper, fault injection and error injection [17] can be identical, and injected faults can also be called injected errors. As shown in Figure 1, we can inject a fault or error to make the function *ioremap_nocache* (line 7937) fail, and let its error handling code (line 8247-8253) executed.

Fault Representativeness. It reflects whether an injected fault can represent a real fault or error to trigger error handling code. If the injected fault is representative, it means that this fault or error can occur in real execution, so the bugs detected in this situation can be regarded as real bugs. Otherwise, the detected bugs are very probably false. Fault representativeness is a key factor, and it decides the effectiveness of SFI [24].

Target Function. A target function is a called function which can fail and trigger error handling code, so it should be fault-injected in SFI. A target function can be a kernel interface or defined in the driver code. *If a target function is real, its failure can be a representative injected fault, because its failure can cause a real error and actually trigger error handling code*. Namely, the realness of target functions largely decides the fault representativeness of SFI. For example in Figure 1, the function *ioremap_nocache* can actually fail and return a null pointer to trigger error handling code, so it is a real target function.

False Positive. There are two kinds of false positives in this paper. One is the false positive of fault representativeness, which is the injected fault that can not actually trigger error handling code. The other is the false positive of bug detection, which is the false detected bug.

Driver Class	Number	"*Goto*" Statement	Return Value
Wireless	116	5109	3757(73.54%)
Ethernet	219	6749	5192(76.93%)
Block	56	1322	1005(76.02%)
Bluetooth	21	121	89(73.56%)
Clock	117	260	213(81.92%)
PCI	51	467	351(75.16%)
USB	268	4148	2971(71.62%)
Total	848	18176	13578(74.70%)

Table 2: Study result of "*goto*" statements.

Driver Class	Number	Error handling	Without Branch
Wireless	116	3903	3111(79.71%)
Ethernet	219	2587	1941(75.03%)
Block	56	149	127(85.23%)
Bluetooth	21	330	239(72.42%)
Clock	117	467	422(90.36%)
PCI	51	470	371(78.94%)
USB	268	701	493(70.32%)
Total	848	8607	6704(77.89%)

Table 3: Study result of branches in error handling code.

3. Characteristics

Previous SFI approaches often have limitations in practical use, such as reporting many false bugs and needing much manual effort. One reason is that they are often used for general software, but neglect key characteristics of device drivers. To improve SFI in testing drivers, we first study the source code of Linux device drivers to find key characteristics of error handling code.

3.1 Function Return Value Trigger

Occasional errors in drivers are often triggered with the function failures, which are reflected as bad return values (null pointers or negative integers of error codes). As shown in Figure 1, when an error occurs in memory mapping, *ioremap_nocache* returns a null pointer. In the example, we find that *error handling code is triggered by a bad function return value*. To know about the proportion of this specific form, we write a program to automatically analyze the source code of 848 Linux (version 3.17.2) device drivers from 7 driver classes. These driver classes are all commonly used, so the study result on them can be applicative to most drivers. In the study, we search for "*goto*" statements in the code, because they are often the entries of error handling code according to the *goto-based* strategy [30]. The result is shown in Table 2. The first column shows the driver class name; the second column shows the number of drivers in each class; the third column shows the number of "*goto*" statements; the fourth column shows the number and proportion of "*goto*" statements in the "*if*" branches of bad function return values.

From Table 2, we find that about 75% of "*goto*" statements are in the "*if*" branches of bad function return values. It indicates that most error handling code in device drivers is triggered by bad function return values. There are two common data types of function return values in device drivers, namely pointer and integer. According to the Linux kernel documentation [44], a null pointer or non-zero integer indicates the operation failure. Moreover, different non-zero integers represent different failure types. For example, -EIO indicates an input/output error and -ENODEV indicates no such device. As for the remaining 25% "*goto*" statements, they are triggered by data failures in the code, such as erroneous data read from registers and bad device states.

3.2 Few Branches

In user-mode applications, error handling code often contains many *if* branches [39]. The main reason is that most user-mode applications are based on *fail-recovery* model. During recovery, error handling code should handle other errors. Therefore, multiple faults need to be injected in user-mode applications to cover most error handling code in runtime testing.

Different from user-mode applications, many device drivers are based on the *fail-stop* model [33]. Namely, when an error occurs, the driver only handles it and prepares to exit, but other errors are never handled at that time. Thus, *there are few if branches in error handling code of device drivers*. To validate this characteristic, we also write a program to automatically analyze the source code of these 848 Linux drivers. In the study, we first filter out all annotations and blank lines, and then count source code lines with and without *if* branches in error handling code. The result is shown in Table 3. The first column shows the driver class name; the second column shows the number of drivers in each class; the third column presents the number of source code lines in error handling code; the fourth column presents the number and proportion of source code lines without *if* branches in error handling code.

From Table 3, we can see that nearly 78% of error handling code is not in *if* branches in these drivers. It indicates that injecting a single fault in each test case is enough to cover most error handling code. This characteristic can help to simplify the complexity of injected faults and improve the efficiency of SFI. The remaining 22% error handling code is in *if* branches because different resource-usage states or device states need to be separately handled in the same error handling code.

This characteristic commonly exists in *fail-stop* drivers. However, some drivers like SATA are based on the *fail-recovery* model, namely they will restart when an error occurs. Thus, many branches are needed to handle the recovery procedure. For these drivers, injecting a single fault is not enough to cover most error handling code. In this paper, we mainly focus on *fail-stop* drivers, because they occupy a large part of existing drivers [34].

3.3 Check Decision

Linux drivers are often implemented in C, so built-in error handling mechanisms (such as "*try-catch*") are not supported. *To check whether an occasional error occur, an if check is often used in the source code*. The *if* statement checks whether the key data is erroneous and decides whether error handling code should be executed. This key data can be a common variable or a function return value. Thus, the characteristic in Section 3.1 can be regarded as an aspect of it. For example in Table 2, all "*goto*" statements triggered by bad function return values are in *if* checks. Particularly, most *if* checks for function return values only check whether the value is a null pointer or non-zero integer (line 7938 in Figure 1 is an example). Namely, different bad function return values are often handled by the same error handling code.

This *if* check decision characteristic is also recommended by the Linux kernel documentation [44]. It can help us inject more representative and efficient faults for SFI. Specifically, we can inject faults in the data checked by these *if* checks, to simulate more realistic errors in device drivers.

```
Procedure: Pattern-based extraction strategy
 1:  func_set := ø; cand_set := ø; fault_set := ø;
 2:  func_set := called functions in normal execution traces;
 3:  foreach func in func_set do
 4:      if GetRetType(func) == integer or pointer then
 5:          AddSet(cand_set, func);
 6:      end if
 7   end foreach
 8:  foreach func in cand_set do
 9:      if func's RetVal is checked by "if" in the driver then
10:          AddSet(fault_set, func);
11:      else if func's RetVal is checked in other drivers then
12:          AddSet(fault_set, func);
13:      else if func's RetVal is specified to be checked then
14:          AddSet(fault_set, func);
15:      end if
16:  end foreach
```

Figure 2: Procedure of extracting target functions.

4. Pattern-based Extraction

The representativeness of injected faults is a key factor of SFI [24]. This property largely determines the accuracy of bug detection and the efficiency of runtime testing. Injecting representative faults can simulate realistic errors to trigger real error handling, so detected bugs are very probably real. Meanwhile, useless test cases are less generated when the injected faults are representative, so the time usage can be largely reduced.

A common strategy is to inject random faults, which has been used in many previous SFI approaches [11, 14, 21, 22]. But some studies [15, 17, 24] have proved this strategy can not well represent real errors, and they also introduces many false positives in bug detection. Because most error handling code in drivers is triggered by bad function return values (in Section 3.1), it is feasible to inject faults in some manually selected target functions which can fail at runtime. This strategy has been used in some previous approaches [7, 32, 43] to test drivers, but it has three problems. Firstly, new target functions should be manually selected when testing a new driver. Secondly, it is hard to assure the selected target functions can actually trigger realistic errors at runtime. Thirdly, many real target functions may be omitted in manual selection.

Based on the characteristics mentioned in Section 3.1 and 3.3, we propose a *pattern-based extraction strategy* to automatically and accurately extract real target functions from the source code. Figure 2 shows the main procedure of this strategy, which consists of two phases.

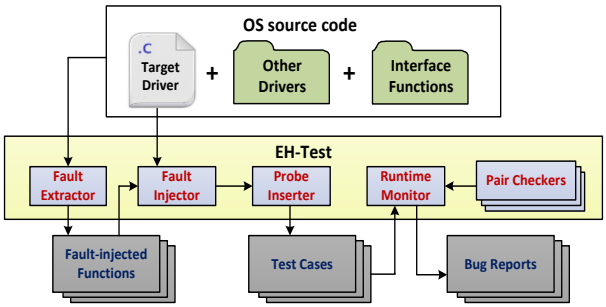


Figure 3: Overall architecture of EH-Test.

Firstly, we run the driver normally on the hardware, and record the runtime traces during normal execution. All functions whose return values are pointers and integers are selected from the recorded runtime traces. These functions are regarded as *candidate functions* for fault injection. Secondly, for each candidate function, we judge whether it can trigger a realistic error. According to the code patterns of Linux device drivers, a candidate function can be regarded as a real target function under three patterns.

Pattern 1. The return value of the candidate function is checked by an *if* statement in the driver. Because an *if* check is often used to decide whether error handling code should be triggered (in Section 3.3). In most cases, this *if* check is often closely behind the function call.

Pattern 2. The return value of the candidate function is checked in other drivers. In some cases, the developer may forget to check the return value of a certain function in the driver. But this function's bad return value can be deemed to trigger a realistic error, when it is checked by an *if* statement in other drivers.

Pattern 3. The return values of some kernel interface functions are clearly specified to be checked in their declarations or annotations, because they can trigger errors. For example in the Linux kernel code, a specific macro "*__must_check*" is defined. If this macro is noted in the declaration of a function, its return value must be checked. The function *pci_request_regions* in Figure 1 uses this macro. Besides, some key phrases in the function annotation also indicate the function return value should be checked. Therefore, the declaration and annotation of candidate functions should be checked as well.

This strategy has three advantages. Firstly, when the driver source code and hardware are available, this strategy can automatically extract target functions without manual effort. Secondly, by using exact runtime information and common code patterns, many unreal target functions are filtered out. Thirdly, no real target functions in the captured runtime traces are omitted. By using this strategy, we can automatically and accurately extract real target functions as representative injected faults to improve the effectiveness of SFI.

5. Approach

To efficiently test error handling code in device drivers, we propose EH-Test based on driver characteristics, code instrumentation and dynamic analysis. Figure 3 shows the overall architecture of EH-Test, which consists of five modules:

- ***Fault extractor.*** This module uses the pattern-based extraction strategy to automatically extract target functions. It needs the source code of the target driver, other drivers and kernel interface functions as input, which can be obtained from the OS source code.
- ***Fault injector.*** This module uses code instrumentation to inject faults by corrupting the return values of target functions. A single fault is injected in each test case.
- ***Probe inserter.*** This module instruments probes in the driver code to collect runtime information and count code coverage during execution. It outputs test cases of the tested driver. Each test case is a loadable driver, which can be directly installed in the operating system.
- ***Runtime monitor.*** This module runs test cases and records the runtime information of the tested driver during execution. It also detects bugs at runtime.
- ***Pair Checkers.*** They are used to check resource usages in drivers. Each pair checker contains the basic information of a pair of resource-acquiring and resource-releasing functions. We have written some pair checkers in EH-Test based on the result of specification mining techniques.

Based on the architecture, two phases are performed when EH-Test works, namely test case generation and runtime testing. The manual work only includes writing pair checkers, checking extracted target functions and rebooting the system when crash bugs are detected.

5.1 Test Case Generation

In this phase, we have two tasks, namely extracting target functions from the code and generating test cases of the driver by injecting faults on target functions. The detailed steps are as follows.

Firstly, we input the driver code and OS source code to the fault extractor. It uses the pattern-based extraction strategy to extract target functions. After extraction, the user also is allowed to check and modify target functions as needed.

Secondly, we inject faults into target functions. A key question is that how many faults should be injected in each test case. Many previous approaches [7, 22, 35, 39] inject multiple faults in each test case, because they aim to cover as much error handling code as possible. But fault scenario explosion may occur in this situation,

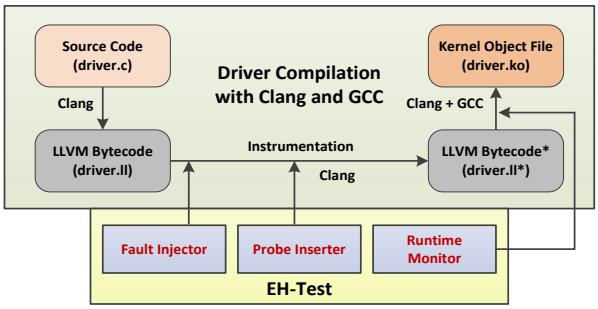


Figure 4: Compilation procedure of the tested driver.

which can largely reduce testing efficiency. To relieve this problem and speed up testing, these approaches have to use some expedients, such as limiting the number of injected faults (or searching paths) [7, 39] and resorting to user guidance [22]. For many Linux drivers, a key characteristic is that there are few *if* branches in error handling code (in Section 3.2). Namely, the error handling code in many device drivers only handles a single error at a time. Thus, to cover most error handling code with less testing time, we only corrupt the return value of one target function in each test case. The target function call is replaced by an *error function* in the code. What this error function does is only returning a bad value. If the return value of the target function is a pointer, the error function will return a null pointer; if the return value of the target function is an integer, the error function will return a random negative number.

Thirdly, we instrument probes to collect runtime information and count code coverage during execution. The runtime information is used to detect bugs in the next phase. The code coverage is used to quantify the effectiveness of runtime testing. Finally, driver test cases are generated. Each test case is a kernel object file, namely a loadable driver.

In the second and third steps, code instrumentation is used. We implement it at compile time using the Clang [40] compiler. Figure 4 shows the compilation procedure of the tested driver. Firstly, we use the Clang compiler to compile the C source code of the driver into the LLVM bytecode. Secondly, we utilize the fault injector and probe inserter to instrument our handled code in the bytecode. Thirdly, we use the Clang compiler to compile the bytecode into the assembly code, and then build the object file using GCC. Finally, we link the object file and the runtime monitor's program together, and generate a kernel object file as a test case.

5.2 Runtime Testing

In this phase, we run each test case on the real hardware and detect bugs during execution. Three kinds of bugs are detected in current implementation, namely crashes, hangs and resource-release omissions.

When driver crashes occur, the OS outputs the dump information into the kernel crash log. Therefore, we can check the kernel crash log to detect and locate crash bugs like null pointer dereferences. For driver hangs, we can detect them by observing whether the system freezes. These two kinds of bugs are easy to observe in real execution.

Function Names	Description	Data
request_irq	Enable / disable the interrupt line and IRQ handling	Para1
free_irq		Para1
pci_enable_device	Initialize / disable the device on the PCI bus	Para1
pci_disable_device		Para1
dma_pool_alloc	Allocate / free a block of consistent memory for DMA	RetVal
dma_pool_free		Para1

Table 4: Selected paired functions in device drivers.

As for resource-release omissions, they are hard-to-find in real execution, because they rarely lead to obvious exceptions. However, they often cause resource-usage problems, such as resource leaks and memory leaks. Moreover, resource-release omissions often occur in device drivers, especially in error handling code [31]. For these reasons, EH-Test should detect resource-release omissions in device drivers. A resource-release omission occurs when a resource-acquiring function is successfully called but its resource-releasing function is not called. For example in Figure 1, *kzalloc* is a resource-acquiring function and it is used to allocate kernel memory, but the resource-releasing function *kfree* is not called, which leads to a resource-release omission. A resource-acquiring function and its resource-releasing function should be called in pairs, so they can be called *paired functions* [20]. Besides, they should operate the same mapped data (parameter or return value) as the handled resource. In EH-Test, we implement some pair checkers to detect resource-release omissions. Each pair checker contains the basic information of a pair of paired functions, including function names and mapped data. Some previous approaches for specification mining [18, 20, 37, 38] can be used to extract paired functions from the code. In this paper, we use the mining result of *PF-Miner* [20], which is a static approach for mining paired functions in Linux drivers, to build the pair checkers. Table 4 shows some selected paired functions in the checkers. The first column shows function names; the second column shows the description; the third column shows the mapped data.

During driver execution, the runtime monitor uses the inserted probes to record the runtime information of function calls and maintains a resource-usage list. For each function call, the monitor checks whether it is in the pair checkers. When a resource-acquiring function is called, the monitor checks its return value to judge

whether the resource is successfully allocated. If it is true, the monitor will create a node containing the function name and mapped data, and add it into the resource-usage list. When a resource-releasing function is called, the monitor scans the list to match the node with the function information. If it is matched, the node will be deleted to indicate the resource is released. When the driver is removed, the monitor checks the nodes in the list. If the list is not empty, it indicates resource-release omissions occur, so the monitor will report them.

6. Evaluation

6.1 Experimental Setup

To validate the effectiveness of EH-Test, we evaluate it on real device drivers. The tested drivers should satisfy three criteria. Firstly, they should be commonly used in practice. Secondly, they should be within the driver classes in Section 3, because they can satisfy the characteristics found by the study. Thirdly, they should run as kernel modules, because the test cases of them can be directly installed and removed without rebooting the operating system. According to these criteria, 15 Linux device drivers are selected, including wireless, USB and Ethernet drivers. Table 5 shows the basic information of tested drivers in Linux 3.17.2.

Class	Driver	Hardware	Lines
Wireless	rtl8180	Realtek RTL8180L Wireless Controller	4.6K
	b43	Broadcom BCM4322 Wireless Controller	57.5K
	iwl4965	Intel 4965AGN Wireless Controller	29.1K
	rt2800	Ralink RT3060 Wireless Controller	22.5K
USB	usb_storage	Kingston 4GB USB disk	7.6K
	uhci_hcd	Intel USB UHCI Controller	7.2K
	ehci_hcd	Intel USB2 EHCI Controller	11.2K
Ethernet	e100	Intel 82559 Ethernet Controller	3.2K
	e1000e	Intel 82572EI Ethernet Controller	28.3K
	igb	Intel 82575EB Ethernet Controller	24.9K
	r8169	Realtek RTL8169 Ethernet Controller	7.4K
	8139too	Realtek RTL8139D Ethernet Controller	2.7K
	3c59x	3Com 3c905B Ethernet Controller	3.4K
	sky2	Marvell 88E8056 Ethernet Controller	7.7K
	ipg	ICPlus IP1000 Ethernet Controller	3.0K

Table 5: Tested drivers in Linux 3.17.2.

The experiment runs on a Lenovo PC with two Intel i5-3470@3.20G processors and 2GB physical memory. GCC 4.8 and Clang 3.2 are used for compilation. We write 75 pair checkers based on the result of *PF-Miner*. For each test case of the drivers, we install it in the system, run it on the workload, and finally remove it. The workload consists of three kinds. For wireless drivers, we turn on WiFi, ping another computer and turn off WiFi; for Ethernet drivers, we ping another computer; For USB drivers, we copy a 4MB file to the USB disk.

Driver	Candidate	Target	Real
rtl8180	39	18	17 (14)
b43	260	55	55 (46)
iwl4965	497	79	74 (64)
rt2800	185	65	57 (48)
usb_storage	60	20	15 (15)
uhci_hcd	120	24	19 (10)
ehci_hcd	160	23	21 (14)
e100	80	33	27 (26)
e1000e	175	62	56 (41)
igb	247	59	51 (51)
r8169	77	15	15 (14)
8139too	64	9	8 (7)
3c59x	59	15	14 (14)
sky2	86	30	25 (25)
ipg	74	16	16 (15)
Total	2183	523	470 (404)

Table 6: Result of the pattern-based extraction.

6.2 Target Function Extraction

The representativeness of injected faults is a key factor of SFI. In this paper, injected faults are bad return values of target functions, and all target functions are automatically extracted by our pattern-based extraction strategy. Thus, the fault representativeness of SFI largely depends on the effectiveness of our pattern-based extraction strategy. There are three important research questions about its effectiveness:

RQ1: *How many unrepresentative candidate functions are automatically filtered out?*
RQ2: *How much is the false positive rate of the strategy?*
RQ3: *How many real target functions are omitted?*

To answer these questions, we first evaluate EH-Test on the 15 drivers to extract candidate functions and target functions. Then we manually check the extracted target functions to judge their realness. Table 6 shows the result in Linux 3.17.2. The first column presents the driver name; the second column shows the number of candidate functions; the third column shows the number of extracted target functions; the fourth column shows the number of real target functions.

From Table 6, we can find that 523 target functions are extracted from 2183 candidate functions. It indicates that 76% candidate functions are automatically filtered out because they are unrepresentative, which can answer RQ1. By manually checking the documents and implementations of the extracted target functions, we find that 470 target functions are real, which means they can actually fail and trigger error handling code. It indicates that the false positive rate of our pattern-based extraction strategy is only 10%, which can answer RQ2. Many false target functions return integers which are also checked by *if* statements, but they reflect different driver configurations or states but never trigger occasional errors. Answering RQ3 is difficult, because target

Driver	Linux 3.1.1					Linux 3.17.2				
	Test case	*Time usage*	*Crash / Hang*	*Resource*	*Bugs*	*Test case*	*Time usage*	*Crash / Hang*	*Resource*	*Bugs*
rtl8180	16	03:14	0 / 0	1 (0)	1	18	04:21	0 / 0	3 (2)	3
b43	62	23:57	0 / 0	1 (1)	1	55	26:34	0 / 0	1 (1)	1
iwl4965	100	36:42	5 / 0	7 (7)	12	79	25:18	5 / 0	8 (8)	13
rt2800	62	19:21	1 / 0	1 (0)	2	65	21:37	0 / 0	1 (0)	1
usb_storage	25	03:35	0 / 0	0 (0)	0	20	03:07	0 / 0	0 (0)	0
uhci_hcd	22	03:47	0 / 0	0 (0)	0	24	03:20	0 / 0	0 (0)	0
ehci_hcd	24	03:50	0 / 0	1 (0)	1	23	03:58	0 / 0	10 (9)	10
e100	33	03:02	1 / 0	0 (0)	1	33	02:28	1 / 0	1 (1)	2
e1000e	66	11:01	0 / 0	0 (0)	0	62	10:30	3 / 0	3 (0)	6
igb	62	10:09	0 / 0	0 (0)	0	59	12:56	0 / 1	6 (6)	7
r8169	15	01:24	0 / 0	0 (0)	0	15	01:43	0 / 0	0 (0)	0
8139too	9	00:45	0 / 0	1 (0)	1	9	00:46	0 / 0	1 (0)	1
3c59x	18	01:26	0 / 0	2 (2)	2	15	01:24	0 / 0	2 (2)	2
sky2	26	01:43	3 / 0	8 (0)	11	30	02:14	4 / 0	0 (0)	4
ipg	17	01:16	0 / 0	0 (0)	0	16	01:28	0 / 0	0 (0)	0
Total	557	125:12	10 / 0	22 (10)	32	523	121:42	13 / 1	36 (29)	50

Table 7: Bug-detection result of EH-Test.

functions are extracted from normal execution traces, but different execution paths may have different runtime traces. Thus, the real target functions within the unexecuted paths will be omitted. However, we find that all target functions in the captured runtime traces are extracted by our strategy.

Reviewing the result, we also find an interesting phenomenon. Most target functions are in the initialization procedure. The data in the parenthesis of the fourth column show the numbers of these functions. They occupy 86% of all target functions. Namely, most kinds of occasional errors in drivers occur in the initialization procedure. In fact, it has been noted in the Linux driver manual [9], and our results can successfully verify it. The explanation for this phenomenon is that different kinds of configurations need to be made in the initialization, and each configuration can cause a kind of occasional error. After the driver is initialized, only several kinds of errors can occur in the running procedure.

6.3 Bug Detection

With the extracted target functions, we perform runtime testing to detect bugs in error handling code. Each test case is generated by making one target function fail. To validate whether EH-Test can find the known bugs having been fixed, we first use EH-Test to test the 15 drivers in an older Linux version 3.1.1 (released in November 2011). Then we test these drivers in a newer Linux version 3.17.2 (released in October 2014) to validate whether EH-Test can find new bugs. Table 7 shows the result. The first column shows the driver name; the second and seventh columns ("*Test case*") show the number of generated test cases; the third and eighth columns ("*Time usage*") present the time usage of the runtime testing; the fourth and ninth columns ("*Crash / Hang*") show the number of detected crashes and hangs; the fifth and tenth columns ("*Resource*") present the number of detected resource-release omissions; the sixth and eleventh columns ("*Bugs*") show the number of detected bugs. Specifically, the number of memory leaks is shown in the parenthesis of the fifth and tenth columns ("*Resource*"), because the memory leak is an important kind of resource-release omission.

From Table 7, we make the following observations:

Firstly, EH-Test finds 32 bugs in the 15 drivers in Linux 3.1.1, including 10 crashes and 22 resource-release omissions. Among these resource-release omissions, 10 are memory leaks. Reviewing the driver code, 8 resource-release omissions (*sky2* driver) and 1 crash (*rt2800* driver) have been fixed in Linux 3.17.2. It indicates that EH-Test can find the known bugs.

Secondly, EH-Test finds 50 bugs in the 15 drivers in Linux 3.17.2, including 13 crashes, 1 hang and 36 resource-release omissions. Among the resource-release omissions, 29 are memory leaks. Moreover, 23 bugs are reserved from the legacy code in 3.1.1, and 27 bugs are introduced due to new implementations. We send all the bugs to the driver developers, and all of them have been confirmed. We also send 17 patches[3] to fix them, and 15 have been applied by the maintainers. It indicates that EH-Test can accurately find new bugs in drivers.

Actually, a threat to validity is that false extracted target functions may introduce false bugs. In our evaluation, no false bugs are detected for this reason.

Thirdly, the time usage of EH-Test is short. About 2 hours are spent in totally testing the 15 drivers, and only several minutes are spent for most drivers. This time usage is shorter than many previous SFI approaches [7, 13, 23] for testing drivers. One reason is that EH-Test uses the pattern-based extraction strategy to filter out many unrepresentative candidate functions, so no redundant test cases are generated. Thus, the test cases are efficient, and the testing time is largely shortened.

[3] The patches can be found in the link: *http://oslab.cs.tsinghua.edu.cn/EHTest/patch.html*

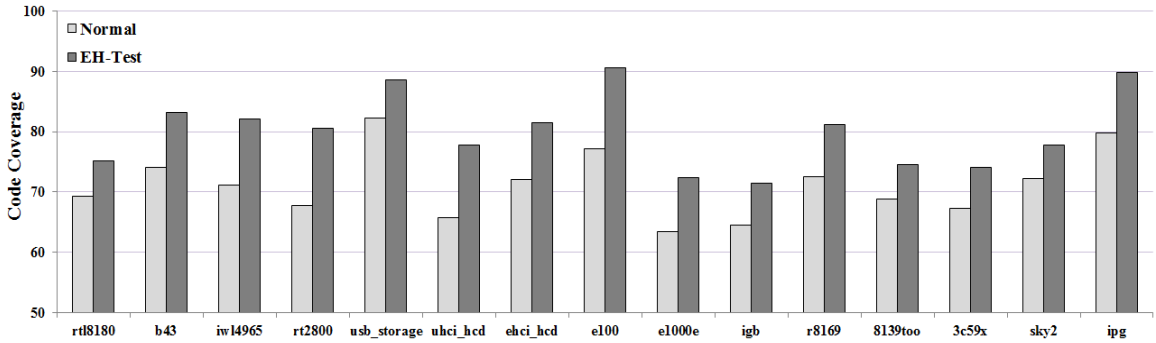


Figure 6: Code coverage of the tested drivers.

Fourthly, resource-release omissions occupy a large part of detected bugs. The main reason is that resource-release omissions often get little attention by developers. The complex execution paths make it difficult to correctly manage resources in error handling code. Meanwhile, resource-release omissions rarely lead to obvious exceptions, so they are hard-to-find in runtime testing.

Figure 5 shows a crash detected by EH-Test in the *e100* driver. The function *pci_pool_create* (line 2967) is called to create a pool of consistent memory blocks for the PCI device, and this function returns a pointer (*nic->cbs_pool*) to this memory area. But the function *pci_pool_create* may fail when the memory is insufficient, and it will return a null pointer in this situation. Thus, a null pointer dereference will occur, when the function *pci_pool_alloc* (line 1910) uses this pointer to allocate a memory block. This crash is detected when we inject a fault in the function *pci_pool_create*. This function is extracted as a target function in our pattern-based extraction strategy, because many other drivers check its return value in the code (pattern 2 in Section 4). To fix this bug, we add an *if* check after the function *pci_pool_create* (line 2967) to check its return value and implement the corresponding error handling code.

Path: linux-3.17.2/drivers/net/ethernet/intel/e100.c

```
1901. static int e100_alloc_cbs(...)
1902. {
          ......
1910.     nic->cbs = pci_pool_alloc(nic->cbs_pool, ...);
1911.     if (!nic->cbs)
1912.         return -ENOMEM;
          ......
1929. }
-----------------------------------------------------
2843. static int e100_probe(....)
2844. {
          ......
2967.     nic->cbs_pool = pci_pool_create(...);
2968.     netif_info(...);
          ......
2990. }
```

Figure 5: A detected bug in the *e100* driver[4].

[4] The applied patch for this bug is in the link: *http://marc.info/?l=linux-netdev&m=143993218231729&w=2*

6.4 Code Coverage

Code coverage is a key criterion in runtime testing. To calculate code coverage, we use the inserted probes to count executed instructions at runtime. Because most target functions are in the initialization procedure, we focus on measuring the code coverage in this procedure. Figure 6 shows the results in Linux 3.17.2. The average code coverage of EH-Test is increased by 8.82% compared to the normal execution. It indicates that hundreds of more instructions are executed in runtime testing.

In fact, not all error handling code can be covered by EH-Test. Firstly, EH-Test only injects faults in target functions, but some error handling code is triggered by erroneous data read from hardware registers. Secondly, our approach injects a single fault in each test case, but some error handling code is triggered by multiple errors. Thirdly, target functions in unexecuted paths are not extracted in our pattern-based strategy, so their error handling code can not be covered. These points cause that the bugs in the uncovered code will be missed.

6.5 Comparison Experiments

Software fault injection and symbolic execution are two runtime techniques which are often used to test drivers.

Software Fault Injection. We compare EH-Test to ADFI [7], a state-of-the-art SFI approach for testing drivers. It uses a bounded trace-based iterative strategy to relieve fault scenario explosion and a permutation-based replay mechanism to assure the fidelity of fault injection. Similar to EH-Test, it injects faults in some target functions and generates test cases to detect bugs. But there are two main differences between ADFI and EH-Test. Firstly, the target functions in ADFI are all manually selected. Only memory, DMA and PCI related interfaces are considered. Thus, much manual work is needed, and many real target functions may be omitted. EH-Test can automatically and accurately extract all target functions in the captured runtime traces without omissions, and the only optional manual work is checking the extracted target functions. Secondly, ADFI injects multiple faults in each test case. The advantage is

that much more configuration and error handling code can be covered to detect more bugs. But numerous test cases are generated, so it spends much more time (often several hours) than EH-Test when testing a driver.

ADFI program and its detailed bug reports are not available, thus we compare the number of its detected bugs from the paper. ADFI and EH-Test both test three drivers with the same workload. For the *e100* and *r8169* drivers, they both find the same number of bugs. For the *ehci_hcd* driver, EH-Test finds 10 bugs, but ADFI does not find any bug in this driver. Reviewing the code, we find that these bugs are triggered by the target functions which are not memory, DMA or PCI related functions, so ADFI omits them.

Symbolic Execution. We select SymDrive [28], a famous symbolic execution approach to make the comparison. This approach uses a symbolic device and some checkers to detect bugs, including memory leaks and null pointer dereferences.

SymDrive program is open-source, and we successfully run it to test the *e100* driver. It runs for nearly 80 minutes and searches 4838 paths, finally exits due to insufficient disk space. In the experiment, SymDrive does not find the bug shown in Figure 5. The reason is that the return value of the function *pci_pool_create* is not marked as a symbolic value in SymDrive, so the corresponding error handling path is not searched. Besides, SymDrive can only test the drivers whose devices are supported by QEMU [4]. But many devices are not supported by the QEMU used in SymDrive, so the drivers for these devices can not be directly tested. The *sky2* and *iwl4965* drivers are the typical examples.

7. Related Work

7.1 Software Fault Injection

In software testing, software fault injection is a technique for testing rarely executed code by deliberately injecting faults. In particular, it is often used to test error handling code in software systems.

Many approaches [11, 14, 21, 22] use random fault injection in software testing. They replace the program data with random faulty data or inject faults into random places, and then run test cases to validate whether the software can properly handle the faults. But random fault injection often leads to poor code coverage and low bug-detection accuracy. To relieve this problem, some approaches [3, 12, 39] use program information to guide fault injection and generate efficient test cases. Moreover, to improve the representativeness of injected faults, much research [10, 17, 24, 25] gives useful solutions through empirical studies. In fact, these approaches are often used for general software, especially user-mode applications. They often neglect the characteristics of device drivers, so their effectiveness may be largely limited when directly testing device drivers.

Besides user-mode applications, SFI is also carefully designed to test drivers [7, 13, 23, 29, 32, 35]. Mendonca et al. [23] perform robustness testing for Windows drivers based on random fault injection. Frequently used kernel interfaces in drivers are called with random parameters. ADFI [7] uses a bounded trace-based iterative strategy to relieve fault scenario explosion and a permutation-based replay mechanism to assure the fidelity of fault injection. But these approaches still have limitations when testing device drivers. A typical limitation is that they often neglect the representativeness of injected faults, and their injected faults are often random or manually selected. To relieve this limitation, EH-Test uses a pattern-based extraction strategy to automatically and accurately extract real target functions as representative injected faults.

7.2 Symbolic Execution

Some approaches [5, 8, 16, 28] introduce symbolic execution in driver testing without the real hardware. DDT [16] is a tool for testing binary drivers against undesired behaviors, such as resource leaks and race conditions. It combines virtualization with selective symbolic execution to test drivers using some modular dynamic checkers. SymDrive [28] provides a symbolic device based on QEMU [4] to simulate real hardware behaviors. It utilizes a favor-success path-selection algorithm in execution, which can increase the exploration priority of executing path at every successful function return within drivers and kernel interfaces. Some checkers are provided to detect common bugs like memory leaks.

Symbolic execution is very time consuming, and it needs much programmer guidance to avoid path explosion. Moreover, many devices can not be simulated well in virtual machines, so their drivers can not be directly tested using symbolic execution.

7.3 Static Analysis

Static analysis is often used to detect bugs in device drivers. It only analyzes the driver source code or binary code without actually running target drivers. Some approaches [1, 2, 6, 26, 27, 36, 41, 42] are based on program verification. For example, SDV [2] is a famous static tool to verify Windows drivers. It abstracts the C source code to a simpler form encoded as a state machine, and checks violations of kernel API usage rules. Some approaches [18, 19, 20, 31] mine implicit specifications from the driver code and detect related bugs. For example, PR-Miner [19] exploits data mining techniques to automatically extract implicit programming rules from software code and detect violations against these extracted rules.

Compared to runtime testing, static analysis lacks precise context information of real execution, so some false positives may be introduced in bug detection.

8. Conclusion

In this paper, we first study the source code of Linux drivers, and find three useful characteristics of error handling code. Then based on these characteristics, we propose a practical approach named EH-Test to efficiently test error handling code and detect bugs inside. It uses a pattern-based extraction strategy to automatically and accurately extract real target functions as representative injected faults. It has been evaluated on 15 Linux drivers and found 50 new real bugs. Our work shows that by introducing the characteristics of target programs, software testing can be more effective.

9. Acknowledgments

We would like to express our gratitude to our shepherd Jon Howell and other reviewers for their helpful comments and suggestions. We also thank the Linux driver developers for their useful feedbacks. This work is supported by Research Grant of Beijing Higher Institution Engineering Research Center and Tsinghua University Initiative Scientific Research Program (2014z09102). Shi-Min Hu is the corresponding author of this paper.

References

[1] S. Amani, L. Ryzhyk, A. F. Donaldson, G. Heiser, A. Legg and Y. Zhu. Static analysis of device drivers: we can do better. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, 2011.

[2] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st European Conference on Computer Systems*, pages 75-88, 2006.

[3] R. Banabic and G. Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th European conference on Computer Systems*, pages 281-294, 2012.

[4] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41-46, 2005.

[5] V. Chipounov, V. Kuznetsov and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265-278, 2011.

[6] E. Clarke, D. Kroening and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168-176, 2004.

[7] K. Cong, L. Lei, Z. Yang and F. Xie. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 361-372, 2015.

[8] K. Cong, F. Xie and L. Lei. Symbolic execution of virtual devices. In *Proceedings of the 13th International Conference on Quality Software*, pages 1-10, 2013.

[9] J. Corbet, A.Rubini and G. K. Hartman, In *Linux Device Drivers*, 3rd ed. O'Reilly Media, pp. 32-35, 2005.

[10] J. A. Duraes and H. S. Madeira. Emulation of software faults: a field data study and a practical approach. In *IEEE Transactions on Software Engineering*, volume 32, issue 11, pages 849-867, 2006.

[11] C. Fu, B. G. Ryder, A. Milanova and D. Wonnacott. Testing of java web services for robustness. In *Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 23-34, 2004.

[12] C. Giuffrida, A. Kuijsten and A. S. Tanenbaum. EDFI: a dependable fault injection tool for dependability benchmarking experiments. In *Proceedings of the 19th Pacific Rim Symposium on Dependable Computing*, pages 31-40, 2013.

[13] A. Johansson, N. Suri and B. Murphy. On the selection of error model(s) for OS robustness evaluation. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 502-511, 2007.

[14] P. Joshi, H. S. Gunawi and K. Sen. PREFAIL: a programmable tool for multiple-failure injection. In *Proceedings of the 2011 International Conference on Object Oriented Programming Systems Languages and Applications*, pages 171-188, 2011.

[15] N. Kikuchi, T. Yoshimura, R. Sakuma and K. Kono. Do injected faults cause real failures? A case study of Linux. In *Proceedings of the 2014 International Symposium on Software Reliability Engineering Workshops*, 2014.

[16] V. Kuznetsov, V. Chipounov and G. Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference*, 2010.

[17] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo and N. Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 397-408, 2014.

[18] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart and G. Muller. WYSIWIB: a declarative approach to finding API protocols and bugs in Linux code. In *Proceedings of the 39th International Conference on Dependable Systems and Networks*, pages 43-52, 2009.

[19] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th International Symposium on Foundations of Software Engineering*, pp. 306-315, 2005.

[20] H. Liu, Y. Wang, L. Jiang and S. Hu. PF-Miner: a new paired functions mining method for Android kernel in error paths. In *Proceedings of the 38th International Computer Software and Applications Conference*, pages 33-42, 2014.

[21] P. D. Marinescu and G. Candea. LFI: a practical and general library-level fault injector. In *Proceedings of the 39th International Conference on Dependable Systems and Networks*, pages 379-388, 2009.

[22] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. In *ACM Transactions on Computer Systems*, volume 29, issue 4, 2011.

[23] M. Mendonca and N. Neves. Robustness testing of the Windows DDK. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 554-564, 2007.

[24] R. Natella, D. Cotroneo, J. A. Duraes and H. S. Maderia. On fault representativeness of software fault injection. In *IEEE Transactions on Software Engineering*, volume 39, issue 1, pages 80-96, 2013.

[25] R. Natella, D. Cotroneo, J. Duraes and H. Maderia. Representativeness analysis of injected software faults in complex software. In *Proceedings of the 40th International Conference on Dependable Systems and Networks*, pages 437-446, 2010.

[26] J. Obdrzalek, J. Slaby and M. Trtik. STANSE: bug-finding framework for C programs. In *Mathematical and Engineering Methods in Computer Science*, volume 7119, pages 167-178, 2012.

[27] H. Post and W. Kuchlin. Integrated static analysis for Linux device driver verification. In *Proceedings of the 6th International Conference on Integrated Formal Methods*, pages 518-537, 2007.

[28] M. J. Renzelmann, A. Kadav and M. M. Swift. SymDrive: testing drivers without devices. In *Proceedings of the 10th International Conference on Operating Systems Design and Implementation*, pages 279-292, 2012.

[29] V. V. Rubanov and E. A. Shatokhin. Runtime verification of Linux kernel modules based on call interception. In *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*, pages 180-189, 2011.

[30] S. Saha, J. Lawall and G. Muller. An approach to improving the structure of error-handling code in the Linux kernel. In *Proceedings of the 2011 International Conference on Languages, Compilers and Tools for Embedded Systems*, pages 41-50, 2011.

[31] S. Saha, J. P. Lozi, G. Thomas, J. L. Lawall and G. Muller. Hector: detecting resource-release omission faults in error-handling code for systems software. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks*, pages 1-12, 2013.

[32] S. D. Shekar, B. B. Meshram and M. P. Varshapriya. Device driver fault simulation using KEDR. In *International Journal of Advanced Research in Computer Engineering and Technology*, pages 580-584, 2012.

[33] M. M. Swift, M. Annamalai, B. N. Bershad and H. M. Leny. Recovering device drivers. In *ACM Transactions on Computer Systems*, volume 24, pages 333-360, 2006.

[34] M. M. Swift, B. N. Bershad and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207-222, 2003.

[35] S. Winter, M. Tretter, B. Sattler and N. Suri. simFI: From single to simultaneous software fault injections. In *Proceedings of the 43rd International Conference on Dependable Systems and Networks*, pages 1-12, 2013.

[36] T. Witkowski, N. Blanc, D. Kroening and G. Weissenbacher. Model checking concurrent linux device drivers. In *Proceedings of the 22nd International Conference on Automated Software Engineering*, pages 501-504, 2007.

[37] Q. Wu, G. Liang, Q. Wang, T. Xie and H. Mei. Iterative mining of resource-releasing specifications. In *Proceedings of the 26th International Conference on Automated Software Engineering*, pages 233-242, 2011.

[38] J. Yang, D. Evans, D. Bhardwaj, T. Bhat and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, pages 282-291, 2006.

[39] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering*, pages 595-605, 2012.

[40] Clang compiler. *http://clang.llvm.org/*

[41] Clang Static Analyzer. *http://clang-analyzer.llvm.org/*

[42] Cppcheck: A tool for static C/C++ code analysis. *http://cppcheck.sourceforge.net/*

[43] Linux Fault Injection Capabilities Infrastructure. *http://www.kernel.org/doc/Documentation/fault-injection/ fault-injection.txt*

[44] Linux Kernel Coding Style. *http://www.kernel.org/doc/Documentation/CodingStyle*

Multicore Locks: The Case is not Closed Yet

Hugo Guiroux[†⋆] Renaud Lachaize[†⋆] Vivien Quéma[†‡⋆]
[†]*Université Grenoble Alpes* [‡]*Grenoble INP*
[⋆]*LIG (CNRS UMR 5217)*

Abstract

NUMA multicore machines are pervasive and many multithreaded applications are suffering from lock contention. To mitigate this issue, application and library developers can choose from the plethora of optimized mutex lock algorithms that have been designed over the past 25 years. Unfortunately, there is currently no broad study of the behavior of these optimized lock algorithms on realistic applications. In this paper, we attempt to fill this gap. We perform a performance study of 27 state-of-the-art mutex lock algorithms on 35 applications. Our study shows that the case is not yet closed regarding locking on multicore machines. Indeed, our conclusions include the following findings: (i) at its optimized contention level, no single lock is the best for more than 52% of the studied workloads; (ii) every lock is harmful for several applications, even if the application parallelism is properly tuned; (iii) for several applications, the best lock changes when varying the number of threads. These findings call for further research on optimized lock algorithms and dynamic adaptation of contention management.

1 Introduction

Today, multicore machines are pervasive and many multithreaded applications are suffering from bottlenecks related to critical sections and their corresponding locks. To mitigate these issues, application and library developers can choose from the plethora of optimized mutex lock algorithms that have been designed over the past 25 years but there is currently no clear study to guide this puzzling choice for realistic applications. In particular, the most recent and comprehensive empirical performance evaluation on multicore synchronization [9], due to its breadth (from hardware protocols to high-level data structures), only provides a partial coverage of locking algorithms. Indeed, the aforementioned study only considers 9 algorithms, does not consider hybrid spinning/blocking waiting policies, omits emerging approaches (e.g., load-control algorithms described in §2) and provides a modest coverage of hierarchical locks [14, 5, 6], a recent and efficient approach. Furthermore, most of the observations are based on microbenchmarks. Besides, in the case of papers that present a new lock algorithm, the empirical observations are often focused on the specific workload characteristics for which the lock was designed [21, 26], or mostly based on microbenchmarks [14, 12].

The present paper provides a broad performance study on Linux/x86 of 27 state-of-the-art mutex lock algorithms on a set of 35 realistic and diverse applications (the PARSEC, Phoenix, SPLASH2 suites, MySQL and an SSL proxy). We make a number of observations, several of which have not been previously mentioned: (*i*) about 60% of the studied applications are significantly impacted by lock performance; (*ii*) no single lock is systematically the best, even for a fixed number of contending cores; (*iii*) worse, at their optimized contention level (individually tuned for each application), the best locks never dominate for more than 52% of the lock-sensitive applications; (*iv*) any of the locks is harmful (i.e., significantly inefficient compared to the best one) for at least several workloads; (*v*) across all the lock-sensitive applications, there is no clear performance hierarchy among the locks, even at a fixed number of contending cores; (*vi*) for a given application, the best lock varies according to both the number of contending cores and the machine; (*vii*) unlike previous recommendations [9] advocating that standard Pthread mutex locks should be avoided for workloads using no more than one thread per core, we find that, with our studied workloads, the current Linux implementation of these locks actually yields good performance for many applications with this pattern. Moreover, we show that all these results hold even when each configuration, i.e., each *(application, lock)* pair, is tuned to its optimal degree of parallelism. From our performance study, we draw two main conclusions. First, specific lock algorithms should not be hardwired into the

code of applications. Second, the observed trends call for further research both regarding lock algorithms and runtime support for parallel performance and contention management.

To conduct our study, manually modifying all the applications in order to retrofit the studied lock algorithms would have been a daunting task. Moreover, using a meta-library that allows plugging different lock algorithms under a common API (such as liblock [26] or libslock [9]) would not have solved the problem, as this would still have required a substantial re-engineering effort for each application. In addition, such meta-libraries provide no or limited support for important features like Pthread condition variables, used within many applications. Therefore, we implemented LiTL[1], a low-overhead library that allows transparent interposition of Pthread mutex lock operations and support for mainstream features like condition variables, without any restriction on the application-level locking discipline.

The remainder of the paper is organized as follows: §2 presents a taxonomy of existing lock designs and the list of algorithms covered by our study. §3 describes our experimental setup and the studied applications. §4 describes the LiTL library. §5 exposes the main results from our empirical observations. §6 discusses related works and §7 concludes the paper.

2 Lock algorithms

2.1 Background

The body of existing works on optimized lock algorithms for multicore architectures is rich and diverse and can be split into the following five categories:

1) Flat approaches correspond to simple algorithms (typically based on one or a few shared variables accessed by atomic instructions) such as: simple spinlock [33], backoff spinlock [2, 30], test and test-and-set (TTAS) lock [2], ticket lock [30], partitioned ticket lock [11], and standard Pthread mutex lock.
2) Queue-based approaches correspond to locks based on a waiting queue in order to improve fairness as well as the memory traffic, such as: MCS [30, 33] and CLH [7, 29, 33].
3) Hierarchical approaches are specifically aimed at providing scalable performance on large-scale NUMA machines, by attempting to reduce the rate of lock migrations among NUMA nodes. This category includes HBO [32], HCLH [28], FC-MCS [13], HMCS [5], AHMCS [6] and the algorithms that stem from the *lock cohorting* framework [14]. A cohort lock is based on a combination of two lock algorithms (similar or different): one used for the global lock and one used for the local locks (there is one local lock per NUMA node); in the usual C-L_A-L_B notation, L_A and L_B respectively correspond to the global and the node-level lock algorithms. The list includes C-BO-MCS, C-PTL-TKT and C-TKT-TKT (also known as Hticket [9]). The *BO*, *PTL* and *TKT* acronyms respectively correspond to backoff lock, partitioned ticket lock, and standard ticket lock.
4) Load-control approaches correspond to algorithms that aim at limiting the number of threads that concurrently attempt to acquire a lock, in order to prevent a performance collapse. These algorithms are derived from queue-based locks. This category includes MCS-TimePub[2] [19] and so-called *Malthusian algorithms* like Malth_Spin and Malth_STP[3] [12].
5) Delegation-based approaches correspond to algorithms in which it is (sometimes or always) necessary for a thread to delegate the execution of a critical section to another thread. The typical benefits expected from such approaches are improved cache locality and better resilience under high lock contention. This category includes Oyama [31], Hendler [20], RCL [26], CC-Synch [15] and DSM-Synch [15].

Another important design dimension is the *waiting policy* used when a thread cannot immediately obtain a requested lock [12]. There are three main approaches: (i) spinning on a memory address, (ii) immediate parking (i.e., blocking the thread) either for a fixed amount of time or until the thread gets a chance to obtain the lock, and (iii) spinning-then-parking (STP), a hybrid strategy using a fixed or adaptive threshold [22]. The choice of the waiting policy is mostly orthogonal to the lock design but, in practice, policies other than pure spinning are only considered for certain types of locks: the queue-based (from categories 2–4 above) and the standard Pthread mutex locks. Besides, note that the GNU C library for Linux provides two versions of Pthread mutex locks: the default one uses parking (via the `futex` syscall) and the second one uses an adaptive spin-then-park strategy. The latter version can be enabled with the `PTHREAD_MUTEX_ADAPTIVE_NP` option [23].

2.2 Studied algorithms

Our choice of studied locks is guided by the decision to focus on *portable* lock algorithms. We therefore exclude the following locks that require strong assumptions on

[1] LiTL: Library for Transparent Lock interposition.

[2] MCS-TimePub is mostly known as MCS-TP but we use MC-TimePub to avoid confusion with MCS_STP.

[3] Malth_Spin and Malth_STP correspond to MCSCR-S and MCSCR-STP, respectively, but we do not use the latter names to avoid confusion with other MCS locks.

Name Total #cores	**A-64** 64	**A-48** 48	**I-48** 48 (no hyperthreading)
Server model	Dell PE R815	Dell PE R815	SuperMicro SS 4048B-TR4FT
Processors Microarchitecture Core clock Last-level cache (per node)	4× AMD Opteron 6272 Bulldozer / Interlagos 2.1 GHz 8 MB	4× AMD Opteron 6344 Piledriver / Abu Dhabi 2.6 GHz 8 MB	4× Intel Xeon E7-4830 v3 Haswell-EX 2.1 GHz 30 MB
Interconnect	HT3 - 6.4 GT/s per link	HT3 - 6.4 GT/s per link	QPI - 8 GT/s per link
Memory #NUMA nodes (#cores/node)	256 GB DDR3 1600 MHz 8 (8)	64 GB DDR3 1600 MHz 8 (6)	256 GB DDR4 2133 MHz 4 (12)
Network interfaces (10 GbE)	2× 2-port Intel 82599	2× 2-port Intel 82599	2-port Intel X540-AT2

Table 1: Hardware characteristics of the testbed platforms.

the application/OS behavior, code modifications, or fragile performance tuning: HCLH, HBO, FC-MCS, and all the delegation-based locks (see Dice et al. [14] for detailed arguments).

Our study considers 27 mutex lock algorithms that are representative of both well-established and state-of-the-art approaches. We use the *_Spin* and *_STP* suffixes to differentiate variants of the same algorithm that only differ in their waiting policy. The *-LS* tag corresponds to optimized algorithms borrowed from libslock [9]. Our set includes ten flat locks (Backoff, Partitioned ticket, Phtread, Pthread adaptive, Spinlock, Spinlock-LS, Ticket, Ticket-LS, TTAS, TTAS-LS), seven queue-based locks (Alock-LS, CLH-LS, CLH_Spin, CLH_STP, MCS-LS, MCS_Spin, MCS_STP), seven hierarchical locks (C-BO-MCS_Spin, C-BO-MCS_STP, C-PTL-TKT, C-TKT-TKT, Hticket-LS, HMCS, AHMCS), and three load-control locks (Malth_Spin, Malth_STP, MCS-TimePub).

3 Experimental setup and methodology

3.1 Testbed and studied applications

Our experimental testbed consists of three Linux-based servers whose main characteristics are summarized in Table 1. All the machines run the Ubuntu 12.04 OS with a 3.17.6 Linux kernel (CFS scheduler), glibc 2.15 and gcc 4.6.3. For our comparative study of lock performance, we consider (i) the applications from the PARSEC benchmark suite (emerging workloads), (ii) the applications from the Phoenix 2 MapReduce benchmark suite, (iii) the applications from the SPLASH2 high-performance computing benchmark suite[4], (iv) the MySQL database running the Cloudstone workload, and (v) SSL proxy, an event-driven SSL endpoint that processes small messages. In order to evaluate the impact of workload changes on locking performance, we also consider so called "long-lived" variants of four of the above workloads denoted with a "_ll" suffix. Note that six of the applications cannot be evaluated on the two 48-core machines because, by design, they only accept a number of threads that correspond to a power of two: facesim, fluidanimate (from PARSEC), fft, ocean_cp, ocean_ncp, radix (from SPLASH2).

Most of these applications use a number of threads equal to the number of cores, except the three following ones: dedup (3× threads), ferret (4× threads) and MySQL (hundreds of threads). Two thirds of the applications use Pthread condition variables.

[4]We excluded the Cholesky application because of extremely short completion times.

3.2 Tuning and experimental methodology

For the lock algorithms that rely on static thresholds, we use the recommended values from the original papers and implementations. The algorithms based on a spin-then-park waiting policy (e.g., Malth_STP [12]) rely on a fixed threshold for the spinning time that corresponds to the duration of a round-trip context switch [22] — in this case, we calibrate the duration using a microbenchmark on the testbed platform.

All the applications are run with memory interleaving (via the `numactl` utility) in order to avoid NUMA memory bottlenecks. Generally, in the experiments presented in this paper, we study the performance impact of a lock for a given contention level, i.e., the number of threads of the application. We vary the contention level at the granularity of a NUMA node (i.e., 8 cores for the A-64 machine, 6 cores for the A-48 machine, and 12 cores for the I-48 machine). For most of the experiments detailed in the paper, the application threads are not pinned to specific cores. The impact of pinning is nonetheless discussed in §5.3.

Finally, each experiment is run at least five times and we compute the average value. Overall, we observe little variability for most configurations. For all experiments, the considered application-level performance metric is the throughput (operations per time unit).

4 The LiTL lock interposition library

In order to carry out the lock comparison study, we have developed LiTL, an interposition library for Linux/x86 allowing transparently replacing the lock algorithm used for Pthread mutexes. We describe its design, implementation, and assess its performance.

4.1 Design

The design of LiTL does not impose any restriction on the level of nested locking and is compatible with arbitrary locking disciplines (e.g., hand-over-hand locking [33]). The pseudo-code of the main wrapper functions of the LiTL library is depicted in Figure 1.

```
// return values and error checks
// omitted for simplification

pthread_mutex_lock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    if (om == null) {
        om = create_and_store_optimized_mutex(m);
    }
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m);
}

pthread_mutex_unlock(pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optmized_mutex(m);
    optimized_mutex_unlock(om);
    real_pthread_mutex_unlock(m);
}

pthread_cond_wait(pthread_cond_t *c,
                  pthread_mutex_t *m) {
    optimized_mutex_t *om = get_optimized_mutex(m);
    optimized_mutex_unlock(om);
    real_pthread_cond_wait(c, m);
    real_pthread_mutex_unlock(m);
    optimized_mutex_lock(om);
    real_pthread_mutex_lock(m);
}

// Note that the pthread_cond_signal and
// pthread_cond_broadcast primitives
// do not need to be interposed
```

Figure 1: Overview of the pseudocode for the main wrapper functions of LiTL.

General principles The primary role of LiTL is to maintain a mapping structure between an instance of the standard Pthread lock (`pthread_mutex_t`) and an instance of the chosen optimized lock type (e.g., MCS_Spin). This implies that LiTL must keep track of the lifecycle of all the application's locks through interposition of the calls to `pthread_mutex_init()` and `pthread_mutex_destroy()`, and that each interposed call to `pthread_mutex_lock()` must trigger a lookup for the instance of the optimized lock. In addition, lock instances that are statically initialized can only be discovered and tracked upon the first invocation of `pthread_mutex_lock()` on them (i.e., a failed lookup leads to the creation of a new mapping).

The `lock`/`unlock` API of several lock algorithms requires an additional parameter (called "struct" hereafter) in addition to the lock pointer. For example, in the case of an MCS lock, this parameter corresponds to the record to be inserted in (or removed from) the lock's waiting queue. In the general case, a struct cannot be reused nor freed before the corresponding lock has been released. For instance, an application may rely on nested critical sections (i.e., a thread T must acquire a lock L_2 while holding another lock L_1). In this case, T must use a distinct struct for L_2 in order to preserve the integrity of L_1's struct. In order to gracefully support the most general cases, LiTL systematically allocates exactly one struct per lock instance and per thread.

Supporting condition variables Dealing with condition variables inside each optimized lock algorithm would be complex and tedious as most locks have not been designed with condition variables in mind. We therefore use the following strategy: our wrapper for `pthread_cond_wait()` internally calls the true `pthread_cond_wait()` function. To issue this call, we need to hold a real Pthread mutex lock (of type `pthread_mutex_t`). This strategy (depicted in the pseudocode of Figure 1) does not introduce high contention on the internal Pthread lock. Indeed, for workloads that do not use condition variables, the Pthread lock is only requested by the holder of the optimized lock associated with the critical section. Furthermore, workloads that use condition variables are unlikely to have more than two threads competing for the Pthread lock: the holder of the optimized lock and a notified thread. Note that the latter claim also holds for workloads that rely on `pthread_cond_broadcast()` because the Linux implementation of this call only wakes up a single thread from the wait queue of the condition variable and directly transfers the remaining threads to the wait queue of the Pthread lock.

Support for specific lock semantics The design of LiTL is compatible with specific lock semantics when the underlying lock algorithms offer the corresponding properties. For example, LiTL supports non-blocking lock requests (`pthread_mutex_trylock()`) for all the currently implemented locks except CLH-based locks and Hticket-LS, which are not compatible with such semantics. Although not yet implemented, LiTL could easily support blocking requests with timeouts for the so-called "abortable" locks (e.g., MCS-Try [34] and MCS-TimePub [19]). Moreover, support for optional Pthread

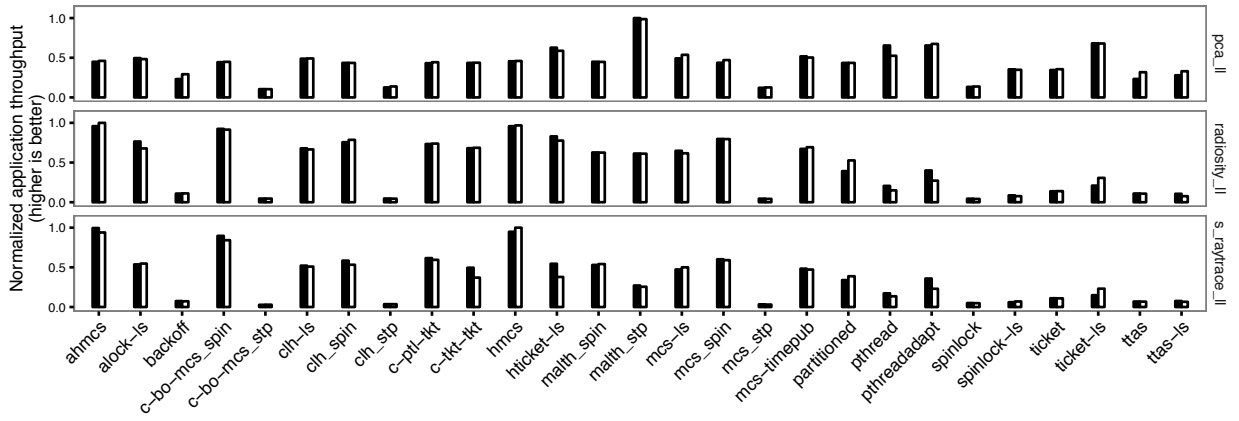


Figure 2: Performance comparison (throughput) of manually implemented locks (black bars) vs. transparently interposed locks using LiTL (white bars). The throughput is normalized with respect to the best performing configuration for a given application (**A-64 machine**).

mutex behavior like reentrance and error checks[5] could be easily integrated in the generic wrapper code by managing fields for the current owner and the lock acquisition counter.

4.2 Implementation

The library relies on a scalable concurrent hash table (CLHT [10]) in order to store, for each Pthread mutex instance used in the application, the corresponding optimized lock instance, and the associated per-thread structs. For well-established locking algorithms like MCS, the code of LiTL borrows from other libraries [9, 1, 26]. Other algorithms are implemented from scratch based on the description of the original papers. For algorithms that are based on a parking or on a spinning-then-parking waiting policy, our implementation directly relies on the `futex` Linux system call.

Finally, the source code of LiTL relies on preprocessor macros rather than function pointers. Indeed, we have observed that the use of function pointers in the critical path introduced a surprisingly high overhead. Moreover, all data structures are cache-aligned in order to mitigate the impact of false sharing.

4.3 Experimental validation

In this section, we assess the performance of LiTL using the A-64 machine. To that end, we compare the performance (throughput) of each lock on a set of applications running in two distinct configurations: manually modified applications and unmodified applications using interposition with LiTL. Clearly, one cannot expect to obtain exactly the same results in both configurations, as the setups differ in several ways, e.g., with respect to the exercised code paths, the process memory layout and the allocation of the locks (e.g., stack- vs. heap-based). However, we show that between both configurations: (i) the achieved performance is close and (ii) the general trends for the different locks remain stable.

We selected three applications: pca_ll, radiosity_ll and s_raytrace_ll. These three applications are particularly lock-intensive and the last two use Pthread condition variables. Therefore, all three represent an unfavorable case for LiTL. Moreover, we focus the discussion on the results under the highest contention level (i.e., when the application uses all the cores of the target machine), as this again represents an unfavorable case for LiTL.

Figure 2 shows the normalized performance (throughput) of both configurations (manual/interposed) for each *(application, lock)* pair: black bars correspond to manually implemented locks, whereas white bars correspond to transparently interposed locks using LiTL. In addition, Table 2 summarizes the performance differences for each application: number of locks for which each version performs better and, in each case, the average gain and the relative standard deviation.

We observe that, for all of the three applications, the results achieved by the two versions of the same lock are very close: the average performance difference is below 5%. Besides, Figure 2 highlights that the general trends observed with the manual versions are preserved with the interposed versions. We thus conclude that using LiTL to study the behavior of lock algorithms in an application yields only very modest differences with respect to the performance behavior of a manually modified version.

[5] Using respectively the `PTHREAD_MUTEX_RECURSIVE` and `PTHREAD_MUTEX_ERRORCHECK` attributes.

		pca_ll	radiosity_ll	s_raytrace_ll
Manual	Winners	10	17	19
	Average Gain	2%	3%	4%
	Rel. Dev.	4%	4%	5%
LiTL	Winners	17	10	8
	Average Gain	2%	3%	3%
	Rel. Dev.	2%	5%	3%

Table 2: Detailed statistics for the performance comparison of manually implemented locks vs. transparently interposed locks using LiTL (**A-64 machine**).

5 Performance study of lock algorithms

In this section, we use LiTL to compare the behavior of the different lock algorithms on different workloads and at different levels of contention. In the interest of space, we do not systematically report the observed standard deviations. However, in order to mitigate the impact of variability, when comparing the performance of two locks, we consider a margin of 5%: lock A is considered better than lock B if B's achieved performance is below 95% of A's. Besides, in order to make fair comparisons, the results presented for the Pthread locks are obtained using the same library interposition mechanism as with the other locks.

Note that some configurations are not tested because of specific restrictions. First, streamcluster, streamcluster_ll, and vips cannot use CLH-based locks or Hticket-LS as they do not support trylocks semantics. Second, we omit the results for most locks with MySQL: given the extremely large ratio of threads to cores, most locks yield performance close to zero. Third, some applications, e.g., dedup and fluidanimate, run out of memory for some configurations.

Finally, for the sake of space, we do not report all the results for the three studied machines. We rather focus on the A-64 machine and provide summaries of the results for the A-48 and I-48 machines. Nevertheless, the entire set of results can be found in a companion technical report [18].

The section is structured as follows. §5.1 provides preliminary observations that drive the study. §5.2 answers the main questions of the study regarding the observed lock behavior. §5.3 discusses additional observations.

5.1 Preliminary observations

Before proceeding with the detailed study, we highlight some important characteristics of the applications.

5.1.1 Selection of lock-sensitive applications

Table 3 shows two metrics for each application and for different numbers of nodes on the A-64 machine: the performance gain of the best lock over the worst one, as well as the relative standard deviation for the performance of the different locks. For the moment, we only focus on the relative standard deviations at the maximum number of nodes (*max nodes*—highest contention) given in the 5th column (the detailed results from this table are discussed in §5.2.1).

We consider that an application is *lock-sensitive* if the relative standard deviation for the performance of the different locks at max nodes is higher than 10% (highlighted in bold font). We observe that about 60% of the applications are impacted by locks. We observe similar trends on the three studied machines (see Table 4).

In the remainder of this study, we focus on lock-sensitive applications.

	Gain 1 node	R.Dev. 1 node	Gain max nodes	R.Dev. max nodes	Gain opt nodes	R.Dev. opt nodes
barnes	10%	2%	36%	8%	31%	7%
blackscholes	11%	2%	2%	1%	2%	1%
bodytrack	1%	0%	9%	2%	4%	1%
canneal	5%	1%	7%	2%	7%	2%
dedup	**683%**	**56%**	**970%**	**55%**	**683%**	**56%**
facesim	**10%**	**2%**	**771%**	**76%**	**14%**	**3%**
ferret	**1%**	**0%**	**349%**	**58%**	**107%**	**25%**
fft	8%	2%	11%	3%	9%	2%
fluidanimate	**48%**	**11%**	**302%**	**28%**	**133%**	**20%**
fmm	**26%**	**7%**	**42%**	**12%**	**42%**	**11%**
freqmine	7%	2%	6%	1%	6%	1%
histogram	7%	2%	20%	5%	12%	3%
kmeans	9%	3%	12%	2%	12%	2%
linear_regression	**9%**	**2%**	**228%**	**22%**	**49%**	**10%**
lu_cb	11%	2%	5%	1%	5%	1%
lu_ncb	17%	5%	8%	2%	8%	2%
matrix_multiply	**7%**	**3%**	**643%**	**51%**	**372%**	**38%**
mysqld	**30%**	**9%**	**174%**	**38%**	**122%**	**34%**
ocean_cp	**17%**	**4%**	**129%**	**15%**	**22%**	**5%**
ocean_ncp	**21%**	**5%**	**118%**	**14%**	**18%**	**4%**
pca	**12%**	**3%**	**358%**	**31%**	**47%**	**8%**
pca_ll	**19%**	**5%**	**665%**	**47%**	**100%**	**20%**
p_raytrace	2%	0%	1%	0%	2%	0%
radiosity	**3%**	**1%**	**91%**	**13%**	**13%**	**4%**
radiosity_ll	**8%**	**2%**	**2299%**	**71%**	**180%**	**29%**
radix	2%	1%	8%	2%	8%	2%
s_raytrace	**4%**	**1%**	**1929%**	**62%**	**126%**	**29%**
s_raytrace_ll	**4%**	**1%**	**3343%**	**79%**	**157%**	**26%**
ssl_proxy	**37%**	**6%**	**1309%**	**63%**	**58%**	**11%**
streamcluster	**13%**	**3%**	**1087%**	**56%**	**13%**	**3%**
streamcluster_ll	**23%**	**4%**	**1305%**	**55%**	**56%**	**12%**
string_match	5%	2%	11%	2%	11%	2%
swaptions	8%	2%	10%	2%	10%	2%
vips	**2%**	**1%**	**334%**	**32%**	**8%**	**2%**
volrend	**7%**	**1%**	**161%**	**21%**	**24%**	**5%**
water_nsquared	**10%**	**2%**	**94%**	**14%**	**94%**	**14%**
water_spatial	**24%**	**5%**	**98%**	**15%**	**96%**	**15%**
word_count	4%	1%	17%	3%	12%	2%
x264	4%	1%	6%	2%	5%	2%

Table 3: For each application, performance gain of the best vs. worst lock and relative standard deviation (**A-64 machine**).

	A-64	A-48	I-48
# tested applications	39	33	33
# lock-sensitive applications	23	19	17

Table 4: Number of tested applications and number of lock-sensitive applications (**all machines**).

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls
dedup	-	252	129	89	95	229	200	204	125	117	75	96	119	119	106	110	113	80	136	120	126	147	118	141	121	145	197
facesim	412	908	425	172	55	888	895	78	460	328	324	379	711	71	1k	948	87	26	895	91	67	726	35	919	462	489	530
ferret	134	176		46		170	174		109	63	100	108	57		194	192			173					182	34		7
fluidanimate	-	72			9	-	-	-				-	7	53	8	12	54	7				16		13	11	6	65
fmm							15	12																			
histogram	95	88	90	95	95	87	92	92	84	79	94	90	90	88	89	85	109	84	89	125	88	107	87	105	102	97	104
linear_regression	44	227	12	21	132	67	45	34	7	49	44	15	25	8	51	47	24		50	10	8	38	8	21			27
matrix_multiply		259								92	287	66			62				7				64		65		55
mysqld	-	-	-	-		-	-	-	-	-	-	-	-		-	-		25	-			-	-	-	-	-	-
ocean_cp	107	97	114	81	70	103	124	121	89	92	96	73	87	75	111	114	82	45	103	72	73	234	49	136	60	106	173
ocean_ncp	93	99	90	73	69	90	93	79	76	90	81	73	84	85	73	92	95	61	98	97	85	206	56	89	57	93	186
pca	77	79	163	42	370	69	44	148	40	34	68	49	37		49	55	134	19	50	97	36	229	80	116	35	160	130
pca_ll	91	81	219	14	582	74	41	321	23	16	88	31	7	21	58	41	403		21	195	114	513	168	108	51	206	476
radiosity																	69						21		10		53
radiosity_ll		12	413		1k	13	10	699	33	19			7		13	11	792	18	48	157	71	987	164	296	97	411	615
s_raytrace		18	185		1k		66	460		14	13	16		7			436		100	88	14	269	50	134	149	195	154
s_raytrace_ll	19	96	781	17	2k	110	107	1k	83	180	15	170	68	161	108	88	1k	118	178	371	185	1k	308	495	301	857	881
ssl_proxy	44	69	695	33	1k	107	61	1k	61	103	608	78	36	52	95	99	1k	73	87	268	195	2k	268	360	139	718	957
streamcluster	2k	2k	4k	2k	2k	-	-	-	1k	2k	1k	-	4k	16k	4k	3k	16k	1k	1k	2k	3k	9k	2k	5k	4k	4k	7k
streamcluster_ll	421	246	829	410	497	-	-	-	266	275	250	-	816	4k	774	590	4k	301	275	446	450	2k	585	1k	615	718	1k
vips	64	56	22	400	32	-	-	-	331	189	131	-	229	18	46	51	18	21	60	20	21	20	23	37	28	22	26
volrend	52	88	97	62	99	72	82	123	50	62	52	59	69	128	79	86	109	82	83	131	162	222	114	74	70	108	154
water_nsquared																											
water_spatial																											

Table 5: For each *(application, lock)* pair, performance gain (in %) of the optimized configuration over the max-node configuration. The background color of a cell indicates the number of nodes (1, 2, 4, 6, or 8 nodes) for the optimized configuration: 1 | 2 | 4 | 6 | 8. Dashes correspond to untested cases. (**A-64 machine**).

5.1.2 Selection of the number of nodes

In multicore applications, optimal performance is not always achieved at the maximum number of available nodes (abbreviated as *max nodes*) due to various kinds of scalability bottlenecks. Therefore, for each *(application, lock)* pair, we empirically determine the *optimized configuration* (abbreviated as *opt nodes*), i.e., the number of nodes that yields the best performance. For the A-64 and A-48 machines, we consider 1, 2, 4, 6, and 8 nodes. For the I-48 machines, we consider 1, 2, 3, and 4 nodes. Note that 6 nodes on A-64 and A-48 correspond to 3 nodes on I-48, i.e., 75% of the available cores.

The results for the A-64 machine are displayed in Table 5. For each *(application, lock)* pair, the corresponding cell indicates the performance gain of the optimized configuration with respect to the max-node configuration. The background color of a cell indicates the number of nodes for the optimized configuration. In addition, Table 6 provides a breakdown of the *(application, lock)* pairs according to their optimized number of nodes for all machines.

We observe that, for many applications, the optimized number of nodes is lower than the max number of nodes. Moreover, we observe (Table 5) that the performance gain of the optimized configuration is often extremely large. This confirms that tuning the degree of parallelism has frequently a very strong impact on performance. We also notice that, for some applications, the optimized number of nodes varies according to the chosen lock.

	A-64	A-48
1 Node	11%	9%
2 Nodes	28%	24%
4 Nodes	27%	21%
6 Nodes	7%	9%
8 Nodes	27%	37%

	I-48
1 Node	33%
2 Nodes	14%
3 Nodes	8%
4 Nodes	45%

Table 6: Breakdown of the *(application, lock)* pairs according to their optimized number of nodes (**all machines**).

In light of the above observations, the main questions investigated in the study (§5.2) will be considered from two complementary angles: (i) comparing locks at a fixed number of nodes, and (ii) comparing locks at their optimized configurations (i.e., with possibly a different number of nodes for each). The first angle offers insight for situations in which the degree of parallelism cannot be adjusted, while the second is useful for scenarios in which more advanced application tuning is possible.

5.2 Main questions

5.2.1 How much do locks impact applications?

Table 3 shows, for each application, the performance gain of the best lock over the worst one at 1 node, max nodes, and opt nodes for the A-64 machine. The table also shows the relative standard deviation for the performance of the different locks.

We observe that the impact of locks on the performance of applications depends on the number of nodes. **At 1 node, the impact of locks on lock-sensitive applications is moderate**. More precisely, most applications exhibit a gain of the best lock over the worst one that is lower than 30%. In contrast, **at max nodes, the impact of locks is very high for all lock-sensitive applications**. More precisely, the gain brought by the best lock over the worst lock ranges from 42% to 3343%. Finally, **at the optimized number of nodes, the impact of locks is high, but noticeably lower than at max nodes.** We explain this difference by the fact that, at max nodes, some of the locks trigger a performance collapse for certain applications (as shown in Table 5), which considerably increases the observed performance gaps between locks. We observe the same trends on the A-48 and I-48 machines (see the companion technical report [18]).

5.2.2 Are some locks always among the best?

Table 7 shows the *coverage* of each lock, i.e., how often it stands as the best one (or is within 5% of the best) over all the studied applications for the A-64 machine. The results are shown for three configurations: 1 node, max nodes, and opt nodes. Besides, Table 8 displays, for each machine (at 1 node, max nodes and opt nodes) the following metrics aggregated over the different locks: the min and max coverage, the average coverage, and the relative standard deviation of the coverage.

	Number of nodes		
Locks	1	Max	Opt
ahmcs	67%	24%	52%
alock-ls	52%	4%	30%
backoff	83%	30%	26%
c-bo-mcs_spin	74%	22%	39%
c-bo-mcs_stp	62%	12%	29%
clh-ls	63%	5%	37%
clh_spin	68%	5%	37%
clh_stp	63%	16%	21%
c-ptl-tkt	57%	22%	35%
c-tkt-tkt	74%	22%	39%
hmcs	65%	22%	48%
hticket-ls	63%	16%	37%
malth_spin	61%	9%	26%
malth_stp	54%	29%	29%
mcs-ls	74%	4%	30%
mcs_spin	70%	22%	48%
mcs_stp	79%	21%	29%
mcs-timepub	54%	38%	29%
partitioned	70%	22%	39%
pthread	50%	21%	29%
pthreadadapt	58%	33%	29%
spinlock	65%	26%	30%
spinlock-ls	57%	30%	35%
ticket	74%	22%	39%
ticket-ls	74%	13%	35%
ttas	83%	26%	43%
ttas-ls	65%	0%	9%

Table 7: For each lock, fraction of the lock-sensitive applications for which the lock yields the best performance for three configurations: 1 node, max nodes, and opt nodes (**A-64 machine**).

# nodes	Coverage	A-64	A-48	I-48
1	[min; max]	[50%; 83%]	[27%; 83%]	[44%; 89%]
	Avg.	66%	66%	62%
	Rel. Dev.	9%	15%	12%
Max	[min; max]	[0%; 38%]	[0%; 42%]	[5%; 50%]
	Avg.	19%	17%	24%
	Rel. Dev.	10%	12%	11%
Opt	[min; max]	[9%; 52%]	[0%; 47%]	[5%; 50%]
	Avg.	34%	21%	28%
	Rel. Dev.	9%	13%	12%

Table 8: Statistics on the coverage of locks for three configurations: 1 node, max nodes, and opt nodes (**all machines**).

We make the following observations (Table 8). **No lock is among the best for more than 89% of the applications at 1 node and for more than 52% of the applications both at max nodes and at the optimal number of nodes**. We also observe that the average coverage is much higher at 1 node than at max nodes, and slightly higher at the optimized number of nodes than at max nodes. This is directly explained by the observations made in §5.2.1. First, at 1 node, locks have a much lower impact on applications than in other configurations and thus yield closer results, which increases their likelihood to be among the best ones. Second, at max nodes, all of the different locks cause, in turn, a performance collapse, which reduces their likelihood to be among the best locks. This latter phenomenon is not observed at the optimized number of nodes. We observe the same trends on the A-48 and I-48 machines (see the companion technical report [18]).

5.2.3 Is there a clear hierarchy between locks?

Table 9 shows pairwise comparisons for all locks, at max nodes on the A-64 machine. In each table, cell $(rowA, colB)$ contains the score of lock A vs. lock B, i.e., the percentage of applications for which lock A is at least 5% better than lock B. For example, Table 9 shows that for 38% of the applications, AHMCS performs at least 5% better than Backoff at the optimized number of nodes. Similarly, the table shows that Backoff is at least 5% better than AHMCS for 29% of the applications. From these two values, we can conclude that the two above mentioned locks perform very closely for 33% of the applications. At the end of each line (resp. column), the table also shows the mean of the fraction of applications for which a lock is better (resp. worse) than others. Besides, the latter two metrics are summarized for the three machines in Table 10.

We observe that **there is no clear global performance hierarchy between locks**. More precisely, for most pairs of locks (A, B), there are some applications for which A is better than B, and vice-versa (Table 9). The only marginal exceptions are the cells having 0% for value. This corresponds to pairs of locks (A, B) for which A

	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	average
ahmcs		19	38	48	29	22	17	61	19	48	5	33	33	43	38	38	48	52	24	38	43	57	48	33	33	43	38	36
alock-ls	19		39	30	26	16	16	58	17	22	9	26	39	30	22	26	43	30	9	39	43	48	39	35	30	35	39	30
backoff	29	35		30	26	37	37	58	26	26	35	32	35	26	35	30	52	30	17	35	39	30	26	4	22	0	39	30
c-bo-mcs_spin	33	48	43		35	37	32	74	22	17	39	32	39	48	39	9	48	13	22	39	39	39	43	48	39	35	65	38
c-bo-mcs_stp	33	43	35	22		42	32	74	17	22	30	21	22	25	26	26	42	21	13	33	33	39	26	26	22	26	61	31
clh-ls	22	21	37	42	32		16	47	26	26	16	26	37	37	16	32	47	26	16	42	47	53	47	47	42	42	47	34
clh_spin	22	32	32	32	26	32		53	21	37	21	42	32	26	32	21	47	32	11	37	37	47	42	32	42	37	47	33
clh_stp	33	32	5	16	11	37	16		26	16	26	26	16	11	21	16	11	5	11	11	11	21	21	11	26	11	32	18
c-ptl-tkt	19	35	35	39	30	32	21	68		26	22	26	26	43	30	26	57	39	17	39	35	48	35	30	30	35	57	35
c-tkt-tkt	24	39	35	26	39	32	26	74	26		30	32	48	65	43	17	57	22	9	39	43	39	43	39	43	35	65	38
hmcs	14	30	39	35	22	42	32	74	17	39		32	39	35	35	26	52	39	26	39	39	48	39	30	30	30	52	36
hticket-ls	17	16	47	32	26	21	32	74	11	21	5		32	42	11	26	53	32	11	42	42	53	42	37	26	47	58	33
malth_spin	14	35	22	22	26	26	16	63	13	17	22	16		22	22	13	39	17	4	35	35	35	39	17	13	17	48	25
malth_stp	24	35	22	35	21	32	37	58	17	17	26	21	4		22	17	33	25	9	33	29	35	22	17	17	17	48	26
mcs-ls	24	17	35	35	35	21	26	63	13	17	17	16	35	26		17	39	17	4	39	43	43	35	30	17	35	48	29
mcs_spin	29	43	35	26	39	37	32	68	26	17	39	47	39	43	43		43	22	22	35	39	35	43	39	30	39	61	37
mcs_stp	29	35	9	22	21	32	32	42	22	9	30	26	17	17	26	9		12	17	21	25	17	17	13	17	13	39	22
mcs-timepub	33	39	35	22	33	42	37	68	17	9	30	32	39	29	22	9	38		13	29	33	30	35	30	30	30	57	32
partitioned	24	39	26	39	43	32	32	68	26	22	39	53	52	43	35	35	61	35		43	48	48	43	26	43	35	65	41
pthread	29	39	22	26	25	37	32	58	22	17	39	26	30	25	35	26	46	25	13		21	39	13	17	13	17	43	28
pthreadadapt	29	43	22	35	21	37	37	53	30	26	35	26	26	25	35	30	42	25	17	21		22	22	17	17	17	43	29
spinlock	29	39	9	26	17	37	32	53	35	13	39	32	43	35	35	22	39	17	22	26	30		26	13	30	9	35	29
spinlock-ls	29	39	26	30	35	26	26	63	26	30	35	16	30	30	30	30	48	30	22	43	30	48		26	13	26	57	33
ticket	29	35	9	26	26	32	32	63	26	22	35	32	30	26	30	26	48	22	13	26	39	30	26		22	0	39	29
ticket-ls	19	22	30	26	39	26	32	68	26	26	22	11	35	39	22	26	52	26	26	35	48	43	39	30		30	52	33
ttas	24	35	4	26	22	37	26	63	26	17	35	32	30	26	30	30	52	17	17	30	35	30	26	4	26		30	28
ttas-ls	19	17	9	17	13	21	16	42	13	13	4	5	22	22	9	22	30	9	13	17	22	30	17	13	4	9		17
average	25	33	27	29	28	32	28	62	22	22	26	28	32	32	29	23	45	25	15	33	36	39	33	26	26	26	49	

Table 9: For each pair of locks *(rowA, colB)* at the optimized number of nodes, score of lock A vs lock B: percentage of applications for which lock A performs at least 5% better than B (**A-64 machine**).

	Better			Worse		
Lock	A-64	A-48	I-48	A-64	A-48	I-48
ahmcs	36%	40%	52%	25%	28%	25%
alock-ls	30%	42%	37%	33%	25%	32%
backoff	30%	29%	23%	27%	33%	45%
c-bo-mcs_spin	38%	47%	46%	29%	25%	15%
c-bo-mcs_stp	31%	25%	38%	28%	44%	25%
clh-ls	34%	46%	32%	32%	32%	38%
clh_spin	33%	38%	33%	28%	34%	37%
clh_stp	18%	11%	8%	62%	72%	71%
c-ptl-tkt	35%	44%	54%	22%	26%	13%
c-tkt-tkt	38%	42%	51%	22%	27%	15%
hmcs	36%	50%	52%	26%	21%	17%
hticket-ls	33%	45%	42%	28%	25%	17%
malth_spin	25%	36%	31%	32%	37%	35%
malth_stp	26%	20%	28%	32%	53%	36%
mcs-ls	29%	43%	35%	29%	22%	26%
mcs_spin	37%	38%	36%	23%	33%	23%
mcs_stp	22%	23%	20%	45%	59%	52%
mcs-timepub	32%	38%	34%	25%	34%	29%
partitioned	41%	42%	38%	15%	32%	23%
pthread	28%	33%	34%	33%	43%	35%
pthreadadapt	29%	34%	34%	36%	38%	36%
spinlock	29%	35%	20%	39%	44%	49%
spinlock-ls	33%	41%	38%	33%	30%	31%
ticket	29%	23%	17%	26%	44%	53%
ticket-ls	33%	40%	28%	26%	24%	35%
ttas	28%	28%	24%	26%	34%	44%
ttas-ls	17%	27%	20%	49%	42%	52%

Table 10: For each lock, at the optimized number of nodes, mean of the fraction of applications for which the lock is better (resp. worse) than other locks (**all machines**).

never yields better performance than *B*. The results at max nodes (not shown due to lack of space) exhibit similar trends as the ones at opt nodes. Besides, we make the same observations (both at opt nodes and max nodes) on the A-48 and I-48 machines (see the companion technical report [18]).

5.2.4 Are all locks potentially harmful?

Our goal is to determine, for each lock, if there are applications for which it yields substantially lower performance than other locks and to quantify the magnitude of such performance gaps. Table 11 displays, for the A-64 machine, the performance gain brought by the best lock with respect to each of the other locks for each application at max nodes (top part) and at the optimized number of nodes for each lock (bottom part). For example, the top part of the table shows that for the dedup application, the best lock (0%, here Spinlock-LS) is 598% better than the Alock-LS lock. The gray cells highlight values greater than 15%. Thus, for each lock in a column, the number of grey cells corresponds to the number of applications for which the lock is beaten by a gap of 15% or more by the best lock(s) for this application. In addition, Table 12 displays, for each machine, the fraction of applications that are significantly hurt by a given lock.

On the three machines, we observe that, **both at max**

Applications	ahmcs	alock-ls	backoff	c-bo-mcs_spin	c-bo-mcs_stp	clh-ls	clh_spin	clh_stp	c-ptl-tkt	c-tkt-tkt	hmcs	hticket-ls	malth_spin	malth_stp	mcs-ls	mcs_spin	mcs_stp	mcs-timepub	partitioned	pthread	pthreadadapt	spinlock	spinlock-ls	ticket	ticket-ls	ttas	ttas-ls	
dedup	-	598	4	135	137	970	575	576	27	11	145	130	130	129	123	127	128	105	14	6	2	2	0	4	0	5	579	
facesim	298	701	323	107	25	680	687	52	333	224	234	273	531	40	771	710	52	0	685	56	44	572	6	719	340	368	409	
ferret	329	297	10	84	0	261	312	0	286	228	255	291	196	0	349	317	0	4	314	0	1	10	0	331	84	9	11	
fluidanimate	-	301	0	57	65	-	-	-	35	14	72	-	36	95	50	40	94	50	14	5	12	26	0	17	15	9	201	
fmm	41	37	15	3	26	38	39	33	30	0	35	32	16	14	32	2	0	0	14	25	23	2	25	15	27	17	34	
histogram	1	2	8	3	4	3	3	12	2	0	2	0	0	1	5	1	14	1	4	19	2	18	3	11	5	8	12	
linear_regression	32	228	24	20	108	57	31	62	0	52	28	11	17	0	49	46	56	3	39	15	0	83	15	32	9	19	49	
matrix_multiply	9	559	5	26	7	18	9	3	24	136	608	642	5	3	639	27	2	0	33	3	3	5	637	3	633	5	630	
mysqld	-	-	-	-	30	-	-	-	-	-	-	-	-	0	-	-	7	173	-	97	102	-	-	-	-	-	-	
ocean_cp	31	18	37	22	16	27	38	38	24	29	29	15	23	27	27	43	32	0	24	11	19	129	5	55	5	38	81	
ocean_ncp	27	28	29	30	9	25	27	28	12	28	16	10	20	22	14	36	37	11	29	31	27	118	0	25	2	29	93	
pca	65	69	155	46	357	61	48	220	40	38	59	39	38	0	43	58	214	23	45	110	39	252	75	110	23	157	112	
pca_ll	47	38	251	24	664	25	51	511	30	24	41	0	18	36	17	50	526	15	27	206	68	584	128	128	17	241	338	
radiosity	14	12	0	0	1	13	9	0	8	1	7	9	9	12	10	1	91	0	1	0	0	1	33	0	19	0	71	
radiosity_ll	0	47	801	9	2k	50	16	2k	35	45	3	28	59	63	62	12	2k	44	76	567	267	2k	396	614	193	825	1k	
s_raytrace	2	24	536	17	2k	9	75	1k	8	27	18	38	26	64	16	0	1k	13	122	230	122	714	118	412	225	554	471	
s_raytrace_ll	6	82	1k	18	3k	96	87	3k	68	169	0	164	84	291	99	69	3k	111	157	639	335	2k	428	813	332	1k	1k	
ssl_proxy	0	18	532	1	1k	47	16	879	9	41	379	20	16	35	43	47	900	29	36	293	153	1k	249	271	85	539	735	
streamcluster	45	24	153	13	63	-	-	-	7	13	3	-	210	1k	183	118	979	6	0	90	133	505	33	290	166	177	395	
streamcluster_ll	61	6	188	20	55	-	-	-	0	17	6	-	234	1k	202	133	1k	34	13	77	102	518	65	263	139	155	411	
vips	41	38	4	333	17	-	-	-	267	145	101	-	177	0	28	28	1	3	37	0	2	3	1	16	8	4	10	
volrend	2	28	41	9	34	16	25	58	1	9	0	6	17	63	22	26	47	24	24	78	104	161	58	24	16	51	92	
water_nsquared	94	48	2	2	9	58	35	35	7	0	14	10	7	6	9	3	2	7	4	6	7	0	6	4	6	4	37	
water_spatial	97	49	2	11	7	63	40	39	4	5	8	4	8	5	5	9	9	10	1	0	0	2	1	1	0	1	41	
dedup	-	378	10	199	193	682	443	436	36	23	237	183	153	152	161	160	158	174	16	16	9	0	10	3	10	3	451	
facesim	2	4	6	0	6	4	4	12	1	0	4	2	2	8	3	1	7	4	3	7	13	7	3	5	3	4	6	
ferret	88	47	6	29	0	37	53	0	89	106	82	92	93	0	56	46	0	3	55	0	0	7	0	56	41	6	7	
fluidanimate	-	133	0	50	51	-	-	-	35	14	64	-	28	27	39	25	26	40	14	5	12	9	0	4	3	3	83	
fmm	41	35	15	3	26	38	21	19	30	0	33	32	16	14	32	2	0	0	14	25	23	1	25	15	27	17	34	
histogram	0	5	9	1	2	6	3	11	6	6	1	1	1	3	6	4	4	5	6	2	3	9	5	3	0	4	5	
linear_regression	2	12	24	11	0	5	1	35	4	14	0	8	5	4	10	11	39	14	4	16	4	48	19	22	15	25	30	
matrix_multiply	9	83	5	22	7	18	9	3	24	23	83	348	5	3	357	23	2	0	24	3	3	5	349	3	343	5	372	
mysqld	-	-	-	-	31	-	-	-	-	-	-	-	-	0	-	-	8	121	-	96	96	-	-	-	-	-	-	
ocean_cp	5	0	7	12	13	4	2	4	10	12	10	11	9	21	0	11	20	14	2	7	15	14	18	9	9	12	10	
ocean_ncp	3	1	6	17	1	3	3	12	0	5	0	0	2	3	3	10	10	8	2	4	7	11	0	4	2	5	5	
pca	2	4	6	13	6	4	12	41	10	12	4	3	11	7	5	12	47	13	6	17	12	17	7	7	0	8	1	
pca_ll	6	5	51	49	54	0	48	100	46	48	3	5	53	55	3	46	71	51	45	43	8	53	17	51	7	53	5	
radiosity	10	9	0	0	1	10	8	0	6	1	7	9	7	10	8	1	13	0	1	0	0	1	10	0	9	0	11	
radiosity_ll	0	31	75	9	53	32	5	180	1	22	3	28	49	59	42	1	165	22	19	159	114	120	88	80	49	80	83	
s_raytrace	2	5	123	16	74	9	5	123	5	11	5	19	26	53	14	0	117	12	10	75	94	120	45	119	30	121	125	
s_raytrace_ll	2	6	79	16	74	7	4	157	5	10	0	11	25	72	9	3	150	11	6	79	74	75	48	75	23	76	78	
ssl_proxy	3	4	17	12	23	5	7	30	0	3	0	0	26	31	9	9	23	11	7	57	27	20	40	19	15	15	16	
streamcluster	11	9	6	0	4	-	-	-	8	1	7	-	10	10	9	1	2	5	7	12	7	2	2	8	8	7	9	
streamcluster_ll	30	29	31	0	9	-	-	-	15	31	28	-	54	47	46	42	39	41	27	36	55	46	2	33	41	31	35	
vips	4	7	3	4	7	-	-	-	3	3	5	-	2	2	5	2	3	3	3	0	1	4	0	2	2	3	5	
volrend	2	4	9	2	2	3	4	8	3	2	0	1	5	8	4	3	7	4	3	17	18	23	12	8	4	10	15	
water_nsquared	94	48	2	2	9	58	35	35	7	0	14	10	7	6	9	3	2	7	4	6	7	0	6	4	6	4	37	
water_spatial	95	49	2	11	7	63	40	39	4	5	8	4	8	5	5	9	9	10	1	0	0	2	1	1	0	1	41	

Max nodes

Opt nodes

Table 11: For each application, at max nodes (top part) and at the optimized number of nodes (bottom part), performance gain (in %) obtained by the best lock(s) with respect to each of the other locks. The grey background highlights cells for which the performance gains are greater than 15%. A line with many gray cells corresponds to an application whose performance is hurt by many locks. A column with many gray cells corresponds to a lock that is outperformed by many other locks. Dashes correspond to untested cases. (**A-64 machine**).

nodes and at the optimal number of nodes, all locks are potentially harmful, yielding sub-optimal performance for a significant number of applications (Table 12). We also notice that locks are significantly less harmful at the optimized number of nodes than at max nodes. This is explained by the fact that several of the locks create performance collapses at max nodes, which does not occur at the optimized number of nodes. Moreover, we observe that, for each lock, the performance gap to the best lock can be significant (Table 11).

5.3 Additional observations

Impact of the number of nodes. Table 13 shows, for each application on the A-64 machine, the number of pairwise changes in the lock performance hierarchy when the number of nodes is modified. For example, in the case of the facesim application, there are 18% of the pairwise performance comparisons between locks that change when moving from a 1-node configuration to a 2-node configuration. Similarly, there are 95% of pairwise comparisons that change at least once when considering

Lock	A-64		A-48		I-48	
	Max	Opt	Max	Opt	Max	Opt
ahmcs	62%	24%	56%	39%	39%	33%
alock-ls	87%	39%	61%	39%	58%	58%
backoff	61%	35%	68%	53%	58%	53%
c-bo-mcs_spin	61%	35%	53%	58%	47%	32%
c-bo-mcs_stp	71%	38%	80%	65%	55%	45%
clh-ls	84%	37%	73%	40%	69%	62%
clh_spin	84%	32%	60%	47%	62%	56%
clh_stp	79%	58%	87%	87%	81%	75%
c-ptl-tkt	52%	30%	53%	42%	47%	26%
c-tkt-tkt	61%	26%	58%	42%	53%	26%
hmcs	61%	26%	37%	37%	37%	16%
hticket-ls	58%	32%	44%	38%	50%	50%
malth_spin	78%	43%	63%	53%	53%	53%
malth_stp	54%	38%	65%	60%	55%	55%
mcs-ls	78%	30%	63%	47%	58%	58%
mcs_spin	70%	26%	63%	53%	58%	58%
mcs_stp	67%	46%	70%	65%	70%	60%
mcs-timepub	42%	25%	65%	55%	50%	50%
partitioned	61%	26%	68%	47%	63%	47%
pthread	62%	50%	60%	55%	60%	55%
pthreadadapt	58%	38%	55%	50%	55%	50%
spinlock	65%	39%	68%	58%	63%	53%
spinlock-ls	57%	39%	58%	42%	58%	47%
ticket	74%	39%	79%	63%	74%	63%
ticket-ls	65%	39%	58%	47%	63%	47%
ttas	61%	35%	68%	53%	63%	58%
ttas-ls	87%	57%	78%	61%	74%	68%

Table 12: For each lock, at max nodes and at the optimized number of nodes, fraction of the applications for which the lock is harmful (**all machines**).

the 1-node, 2-node, 4-node and 8-node configurations.

We observe that, **for all applications, the lock performance hierarchy changes significantly according to the chosen number of nodes**. Moreover, we observe the same trends on the A-48 and I-48 machines (see the companion technical report [18]).

Applications	% of pairwise changes between configurations			
	1/2	2/4	4/8	1/2/4/8
dedup	16%	6%	12%	19%
facesim	18%	38%	81%	95%
ferret	0%	74%	26%	87%
fluidanimate	5%	6%	24%	32%
fmm	33%	10%	19%	45%
histogram	19%	32%	24%	55%
linear_regression	58%	40%	57%	95%
matrix_multiply	16%	27%	45%	54%
mysqld	33%	20%	7%	40%
ocean_cp	54%	53%	72%	94%
ocean_ncp	52%	54%	56%	86%
pca	44%	60%	29%	89%
pca_ll	31%	38%	23%	73%
radiosity	11%	49%	65%	83%
radiosity_ll	66%	28%	14%	92%
s_raytrace	1%	70%	32%	96%
s_raytrace_ll	21%	69%	24%	99%
ssl_proxy	62%	12%	21%	78%
streamcluster	68%	21%	32%	88%
streamcluster_ll	60%	28%	31%	90%
vips	2%	3%	82%	82%
volrend	16%	27%	44%	85%
water_nsquared	23%	24%	13%	52%
water_spatial	12%	10%	10%	29%

Table 13: For each application, percentage of pairwise changes in the lock performance hierarchy when changing the number of nodes (**A-64 machine**).

Impact of the machine. Table 14 shows the number of pairwise lock inversions observed between the machines (both at max nodes and at the optimized number of nodes). More precisely, for a given application at a given node configuration, we check whether two locks are in the same order or not on the target machines.

We observe that **the lock performance hierarchy changes significantly according to the chosen machine**. Interestingly, we observe that there is approximately the same number of inversions between each pair of machines.

# nodes	A-64 vs. A-48	A-48 vs. I-48	A-64 vs. I-48
Max	38%	36%	38%
Opt	30%	29%	31%

Table 14: For each pair of machines, at max nodes and at opt nodes, percentage of pairwise changes in the lock performance hierarchy (**all machines**).

A note on Phtread locks. The various results presented in this paper show that the current Linux **Pthread locks perform well (i.e., are among the best locks) for a significant share of the studied applications**, thus providing a different insight than recent results, which were mostly based on synthetic workloads [9]. Beyond the changes of workloads, these differences may also be explained by the continuous refinement of the Linux Pthread implementation. It is nevertheless important to note that on each machine, some locks stand out as the best ones for a higher fraction of the applications than Pthread locks. Finally, we note that Pthread adaptive locks perform slightly better than standard Pthread locks.

Impact of thread pinning. As explained in §3.2, all the above-described experiments were run without any restriction on the placement of threads, leaving the corresponding decisions to the Linux scheduler. However, in order to better control CPU allocation and improve locality, some developers and system administrators use pinning to explicitly restrict the placement of each thread to one or several core(s). The impact of thread pinning may vary greatly according to workloads and can yield both positive and negative effects [9, 27]. In order to assess the generality of our observations, we also performed the complete set of experiments with an alternative configuration in which each thread is pinned to a given node, leaving the scheduler free to place the thread among the cores of the node. Note that for an experiment with a N-node configuration, the complete application runs on exactly first N nodes of the machine. We chose thread-to-node pinning rather than thread-to-core pinning because we observed that the former generally provided better

performance for our studied applications, especially the ones using more threads than cores. The detailed results of our experiments with thread-to-node pinning are available in the companion technical report [18]. Overall, we observe that **all the conclusions presented in the paper still hold with per-node thread pinning**.

6 Related work

The design and implementation of the LiTL lock library borrows code and ideas from previous open-source toolkits that provide application developers with a set of optimized implementations for some of the most-established lock algorithms: Concurrency Kit [1], liblock [25, 24, 26], and libslock [9]. All of these toolkits require potentially tedious source code modifications in the target applications, even in the case of algorithms that have been specifically designed to lower this burden [3, 33, 36]. Moreover, among the above works, none of them provides a simple and generic solution for supporting Pthread condition variables. The authors of liblock [26] have proposed an approach but we discovered that it suffers from liveness hazards due to a race condition. Indeed, when a thread T calls `pthread_cond_wait()`, it is not guaranteed that the two steps (releasing the lock and blocking the thread) are always executed atomically. Thus, a wake-up notification issued by another thread may get interleaved between the two steps and T may remain indefinitely blocked.

Several research works have leveraged library interposition to compare different locking algorithms on legacy applications (e.g., Johnson et al. [21] and Dice et al. [14]) but, to the best of our knowledge, they have not publicly documented the design challenges to support arbitrary application patterns, nor disclosed the corresponding source code and the overhead of their interposition library has not been discussed.

Several studies have compared the performance of different multicore lock algorithms, either from a theoretical angle or based on experimental results [4, 33, 9, 24, 14]. In comparison, our study encompasses significantly more lock algorithms and waiting policies. Moreover, the bulk of these studies is mainly focused on characterization microbenchmarks while we focus instead on workloads designed to mimic real applications. Two noticeable exceptions are the work from Boyd-Wickizer et al. [4] and Lozi et al. [26] but they do not consider the same context as our study. The former is focused on kernel-level locking bottlenecks, and the latter is focused on applications in which only one or a few heavily contended critical sections have been optimized (after a profiling phase). For all these reasons, we make observations that are significantly different from the ones based on all the above-mentioned studies. Other synchronization-related studies like the one from Gramoli [16] have a different scope and focus on concurrent data structures, possibly based on other facilities than locks.

Finally, some tools have been proposed to facilitate the identification of locking bottlenecks in applications [35, 8, 26]. These publications are orthogonal to our work. We note that, among them, the profilers based on library interposition can be stacked on top of LiTL.

7 Conclusion and future work

Optimized lock algorithms for multicore machines are abundant. However, there are currently no clear guidelines and methodologies helping developers to select the right lock for their workloads. In this paper, we have presented a broad study of 27 locks algorithms with 35 applications on Linux/x86. To perform that study, we have implemented LiTL, an interposition library allowing the transparent replacement of lock algorithms used for Pthread mutex locks. From our study, we draw several conclusions, including the following ones: at its optimized contention level, no single lock dominates for more than 52% of the lock-sensitive applications; any of the locks is harmful for at least several applications; for a given application, the best lock varies according to both the number of contending cores and the machine that executes the application. These observations call for further research on optimized lock algorithms, as well as tools and dynamic approaches to better understand and control their behavior.

The source code of LiTL and the data sets of our experimental results are available online [17].

Acknowledgments

We thank the anonymous reviewers and our shepherd, Tim Harris, for their insightful comments on ealier drafts of this paper. Dave Dice provided detailed answers for our questions on Malthusian locks. Baptiste Lepers provided valuable insights for some of the case studies. Pierre Neyron provided his help to set up experiments on the I-48 machine. Finally, this work has been partially supported by: LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), EmSoc Replicanos and AGIR CAEC projects of Université Grenoble-Alpes and GrenobleINP, and the INRIA/LIG Digitalis project.

References

[1] AL BAHRA, S. Concurrency Kit, 2015. `http://concurrencykit.org`.

[2] ANDERSON, T. E. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transaction on Parallel and Distributed Systems* (Jan. 1990), 6–16.

[3] AUSLANDER, M., EDELSOHN, D., KRIEGER, O., ROSENBURG, B., AND WISNIEWSKI, R. Enhancement to the MCS Lock for Increased Functionality and Improved Programmability. U.S. Patent Application Number 20030200457 (abandoned), October 2003.

[4] BOYD-WICKIZER, S., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. Non-scalable Locks are Dangerous. In *Proceedings of the Linux Symposium* (Ottawa, Canada, July 2012).

[5] CHABBI, M., FAGAN, M., AND MELLOR-CRUMMEY, J. High Performance Locks for Multi-level NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)* (2015), ACM.

[6] CHABBI, M., AND MELLOR-CRUMMEY, J. Contention-conscious, Locality-preserving Locks. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)* (2016), ACM.

[7] CRAIG, T. S. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Tech. Rep. TR 93-02-02, University of Washington, 1993.

[8] DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (2014), OOPSLA '14, ACM.

[9] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)* (2013), ACM.

[10] DAVID, T., GUERRAOUI, R., AND TRIGONAKIS, V. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)* (2015), ACM.

[11] DICE, D. Brief Announcement: A Partitioned Ticket Lock. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)* (2011), ACM.

[12] DICE, D. Malthusian Locks, november 2015. `http://arxiv.org/abs/1511.06035`.

[13] DICE, D., MARATHE, V. J., AND SHAVIT, N. Flat-Combining NUMA Locks. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)* (2011), ACM.

[14] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Transactions on Parallel Computing 1*, 2 (Feb. 2015), 13:1–13:42.

[15] FATOUROU, P., AND KALLIMANIS, N. D. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)* (2012), ACM.

[16] GRAMOLI, V. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)* (2015), ACM.

[17] GUIROUX, H., LACHAIZE, R., AND QUÉMA, V. LiTL source code and data sets, 2016. `https://github.com/multicore-locks`.

[18] GUIROUX, H., LACHAIZE, R., AND QUÉMA, V. Multicore Locks: the Case is not Closed Yet. Technical report, 2016. Available from `https://github.com/multicore-locks`.

[19] HE, B., SCHERER, W. N., AND SCOTT, M. L. Preemption Adaptivity in Time-published Queue-based Spin Locks. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC'05)* (2005), Springer-Verlag.

[20] HENDLER, D., INCZE, I., SHAVIT, N., AND TZAFRIR, M. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)* (2010), ACM.

[21] JOHNSON, F. R., STOICA, R., AILAMAKI, A., AND MOWRY, T. C. Decoupling Contention Management from Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)* (2010), ACM.

[22] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP'91)* (1991), ACM.

[23] KYLHEKU, K. What is PTHREAD_MUTEX_ADAPTIVE_NP?, 2014. `http://stackoverflow.com/a/25168942`.

[24] LOZI, J.-P. *Towards More Scalable Mutual Exclusion for Multicore Architectures*. PhD thesis, UPMC, Paris, July 2014. `http://www.i3s.unice.fr/~jplozi/documents/lozi-phd-thesis.pdf`.

[25] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), USENIX Association.

[26] LOZI, J.-P., DAVID, F., THOMAS, G., LAWALL, J., AND MULLER, G. Fast and Portable Locking for Multicore Architectures. *ACM Transactions on Computer Systems 33*, 4 (Jan. 2016), 13:1–13:62.

[27] LOZI, J.-P., LEPERS, B., FUNSTON, J., GAUD, F., QUÉMA, V., AND FEDOROVA, A. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)* (2016), ACM.

[28] LUCHANGCO, V., NUSSBAUM, D., AND SHAVIT, N. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing (Euro-Par'06)* (2006), Springer-Verlag.

[29] MAGNUSSON, P. S., LANDIN, A., AND HAGERSTEN, E. Queue Locks on Cache Coherent Multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing* (1994), IEEE Computer Society.

[30] MELLOR-CRUMMEY, J. M., AND SCOTT, M. L. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (Feb. 1991), 21–65.

[31] OYAMA, Y., TAURA, K., AND YONEZAWA, A. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing For Symbolic And Irregular Applications (PDSIA'99)* (1999), World Scientific.

[32] RADOVIC, Z., AND HAGERSTEN, E. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)* (2003), IEEE Computer Society.

[33] SCOTT, M. L. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013.

[34] SCOTT, M. L., AND SCHERER, W. N. Scalable Queue-based Spin Locks with Timeout. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP'01)* (2001), ACM.

[35] TALLENT, N. R., MELLOR-CRUMMEY, J. M., AND PORTERFIELD, A. Analyzing Lock Contention in Multithreaded Applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)* (2010), ACM.

[36] WANG, T., CHABBI, M., AND KIMURA, H. Be My Guest — MCS Lock Now Welcomes Guests. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'16)* (2016), ACM.