USENIX Association

Proceedings of USENIX ATC ’14:
2014 USENIX Annual Technical Conference

June 19–20, 2014
Philadelphia, PA

Conference Organizers

Program Co-Chairs

Garth Gibson, *Carnegie Mellon University*

Nickolai Zeldovich, *Massachusetts Institute of Technology*

Program Committee

Muli Ben-Yehuda, *Stratoscale and Technion*

David Brumley, *Carnegie Mellon University*

Haibo Chen, *Shanghai Jiao Tong University*

Adam Chlipala, *Massachusetts Institute of Technology*

Alan L. Cox, *Rice University*

Steven Gribble, *University of Washington and Google*

Steven Hand, *Microsoft Research*

Jon Howell, *Microsoft Research*

Anthony D. Joseph, *University of California, Berkeley*

Terence Kelly, *HP Labs*

Eddie Kohler, *Harvard University*

Jinyang Li, *New York University*

Shan Lu, *University of Wisconsin—Madison*

Pradeep Padala, *VMware*

Vivek Pai, *Princeton University*

KyoungSoo Park, *KAIST*

Dave Presotto, *Google*

Benjamin Reed, *Facebook*

Erik Riedel, *EMC*

Luigi Rizzo, *Università di Pisa*

Mendel Rosenblum, *Stanford University*

Kai Shen, *University of Rochester*

Dilma Da Silva, *Qualcomm Research*

Christopher Small, *Philo Inc.*

Keith A. Smith, *NetApp*

Christopher Stewart, *Ohio State University*

Kobus Van der Merwe, *University of Utah*

Robert N.M. Watson, *University of Cambridge*

Emmett Witchel, *The University of Texas at Austin*

Ding Yuan, *University of Toronto*

External Reviewers

Samy Al Bahra

Kayvon Fatahalian

Anne Holler

K.R. Jayaram

Pramod Joisha

Alan Karp

Ishan Mitra

Brad Morrey

Derek Murray

Stan Park

Ohad Rodeh

Abhinav Shrivastava

Gokul Soundararajan

Kiran Srinivasan

Michael Swift

Haris Volos

Yin Wang

USENIX ATC ’14:
2014 USENIX Annual Technical Conference
June 19–20, 2014
Philadelphia, PA

Thursday, June 19

Big Data

Virtualization

(Thursday, June 19, continues on p. iv)

Storage

Hardware and Low-level Techniques

Friday, June 20, 2014

Distributed Systems

Networking

Security and Correctness

(Friday, June 20, continues on p. vi)

Flash

Message from the 2014 USENIX Annual Technical Conference Program Co-Chairs

Welcome to the 2014 USENIX Annual Technical Conference.

This year's program committee has put together a program of 44 papers, including five short papers. These papers span a wide range of topics covering both novel research contributions and practical ideas in storage systems, networking, big data, distributed systems, security, virtualization, multi-core systems, and hardware.

We received a record number of submissions this year. Authors submitted a total of 245 papers (after registering 305 abstracts a few days before the submission deadline). Forty-nine of these were submissions of short papers, which had to be at most six pages long, and the other 196 were full-length papers that had to be at most 12 pages long. The program co-chairs rejected four of the full-length papers without review for violating format requirements (all were judged to give the authors at least a column of additional text).

The program committee reviewed the submissions over two rounds. In the first round, each of the 241 papers received two reviews. Papers receiving an "accept" or "strong accept" review moved on to the second round, as well as papers with two "weak accept" reviews, papers without sufficiently confident reviewers, and papers where reviewers explicitly said the paper should advance to the second round. This amounted to a total of 110 papers. The remaining 131 papers were tentatively rejected. Papers in the second round received three more reviews, as well as additional reviews from external experts. Altogether, this produced 834 reviews.

After an online discussion among reviewers, the program committee met in person in April in Seattle, immediately after the USENIX NSDI conference; five of the members were unable to attend in person. Over a period of nine hours across two days, the committee discussed 66 papers that were highly ranked or merited further consideration after online discussion, and they decided to accept 36 papers, including four short papers. Shepherds were assigned to specific papers judged to have specific shortcomings.

This year, we introduced a resubmission process: for papers that appeared to contain interesting ideas and techniques but that could not be accepted in their submitted form, the program committee could reject the paper but give the authors an option to resubmit a revised version of the paper. The program committee decided to give 11 rejected papers (including one short paper) this option; each such paper was assigned a contact who helped the authors understand reviewers' concerns. All 11 authors took advantage of this option. The original reviewers read and commented on each resubmitted version over a period of a week and decided to accept 8 out of 11 resubmissions. Out of an average of five original reviewers per resubmitted paper, an average of four engaged in evaluating the resubmitted version.

The committee was comprised of 30 members, plus the two co-chairs. Twelve of them were affiliated with industrial organizations, and 22 were affiliated with academic institutions (two fell in both categories). Program committee members were allowed to submit papers. We followed conventional rules for handling conflicts of interest: conflicted members (or co-chairs) left the room during discussion of conflicted papers. One paper, authored by a co-chair, was separately handled by the other co-chair.

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

Garth Gibson, *Carnegie Mellon University*
Nickolai Zeldovich, *Massachusetts Institute of Technology*
USENIX ATC '14 Program Co-Chairs

ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters*

Faraz Ahmad[*,†], Srimat T. Chakradhar[‡], Anand Raghunathan[†], T. N. Vijaykumar[†]
[*]Teradata Aster, San Carlos, CA, USA, [‡]NEC Laboratories America, Princeton, NJ, USA
[†]School of Electrical and Computer Engineering, Purdue University, IN, USA
faraz.ahmad@teradata.com, chak@nec-labs.com, {raghunathan,vijay}@ecn.purdue.edu

Abstract

MapReduce clusters are usually multi-tenant (i.e., shared among multiple users and jobs) for improving cost and utilization. The performance of jobs in a multi-tenant MapReduce cluster is greatly impacted by the all-Map-to-all-Reduce communication, or Shuffle, which saturates the cluster's hard-to-scale network bisection bandwidth. Previous schedulers optimize Map input locality but do not consider the Shuffle, which is often the dominant source of traffic in MapReduce clusters.

We propose *ShuffleWatcher*, a new multi-tenant MapReduce scheduler that shapes and reduces Shuffle traffic to improve cluster performance (throughput and job turn-around times), while operating within specified fairness constraints. ShuffleWatcher employs three key techniques. First, it curbs intra-job Map-Shuffle concurrency to shape Shuffle traffic by delaying or elongating a job's Shuffle based on the network load. Second, it exploits the reduced intra-job concurrency and the flexibility engendered by the replication of Map input data for fault tolerance to preferentially assign a job's Map tasks to localize the Map output to as few nodes as possible. Third, it exploits localized Map output and delayed Shuffle to reduce the Shuffle traffic by preferentially assigning a job's Reduce tasks to the nodes containing its Map output. ShuffleWatcher leverages opportunities that are unique to multi-tenancy, such overlapping Map with Shuffle across jobs rather than within a job, and trading-off intra-job concurrency for reduced Shuffle traffic. On a 100-node Amazon EC2 cluster running Hadoop, ShuffleWatcher improves cluster throughput by 39-46% and job turn-around times by 27-32% over three state-of-the-art schedulers.

1 Introduction

MapReduce frameworks are commonly used to process large volumes of data on clusters of commodity computers. MapReduce provides easy programmability, automatic data parallelization and transparent fault tolerance [13]. For cost-effectiveness and better utilization, MapReduce clusters are frequently *multi-tenant* (i.e., shared among multiple users and jobs).

The performance of MapReduce clusters is greatly affected by the Shuffle, an all-Map-to-all-Reduce communication, which stresses the network bisection bandwidth. Typical MapReduce workloads contain a significant fraction of Shuffle-heavy jobs (e.g., 60% and 20% of the jobs on the Yahoo and Facebook clusters, respectively, are reported to be Shuffle-heavy [9,39]). Shuffle-heavy MapReduce jobs typically process more data in the Shuffle and Reduce phases and hence run much longer than Shuffle-light jobs [2,3]. As such, Shuffle-heavy jobs significantly impact the cluster throughput. The execution of multiple, concurrent Shuffles due to multi-tenancy worsens the pressure on the network bisection bandwidth. While network switch and *link* bandwidth scale with hardware technology, *bisection* bandwidth is a global resource that is hard to scale up with the cluster's compute and storage resources (CPU, memory, disk). Even with recent advances in data center networks [4], large clusters are typically provisioned for *per-node* bisection bandwidth that is 5-20 times lower than the within-rack bandwidth [13,17,35,37,39].

Several previous multi-tenant schedulers address the problem of fairness among users or jobs (e.g., FIFO [21], Capacity [30], Fair [31] and Dominant Resource Fairness [16] schedulers). Other efforts improve cluster throughput by optimizing data locality in the Map phase [25,39] but do not address Shuffle, which is the dominant source of network traffic in MapReduce. Our goal is to improve performance (cluster throughput and job turn-around time) within specified fairness constraints addressing the Shuffle bottleneck. Recent efforts [10,12,32] propose techniques to manage data center network traffic without changing network load. In contrast, we actively shape and reduce the network load.

We propose ShuffleWatcher, a new multi-tenant MapReduce scheduler that improves performance by exploiting a key trade-off between intra-job concurrency and Shuffle locality. Previous multi-tenant schedulers adopt the approach of maximizing intra-job concurrency, while ensuring a fair division of resources among users. This approach is a carryover from single-tenant scheduling; when there is a single job, utilizing the entire cluster (highest intra-job concurrency) typically ensures fastest turn-around times. Hence, concurrency is not traded for locality (although locality optimizations without sacrificing concurrency are welcome). However, adopting this concurrency-centric scheduling approach in multi-tenancy is neither neces-

[*]work was done while Faraz Ahmad was at Purdue University.

sary (because concurrency may be exploited either within or across jobs), nor beneficial (because it often results in multiple concurrent, contending Shuffles, which saturate the network and degrade throughput).

ShuffleWatcher employs three key mechanisms that leverage the aforementioned trade-off. The first mechanism, called *Network-Aware Shuffle Scheduling (NASS)*, curbs intra-job concurrency at high network loads to shape the Shuffle traffic. Previous schedulers typically overlap a job's Shuffle with its own Map phase by creating and scheduling the Reduce tasks early in the Map phase. This rigidity in Reduce scheduling often results in multiple Shuffle-heavy jobs being concurrently scheduled, thereby saturating the network bisection bandwidth and hurting performance. We make the key observation that multi-tenancy presents a new degree of freedom to overlap the Shuffle and Map across jobs, rather than within a job. Accordingly, NASS curbs the intra-job Map-Shuffle concurrency at high network loads by delaying or elongating a job's Shuffle. This profitably shapes the network traffic to alleviate congestion, while still achieving Map-Shuffle overlap across jobs. To maintain fairness for the user whose Shuffle is delayed, ShuffleWatcher schedules tasks (from the same or another job**)** of the same user that do not stress the network. ShuffleWatcher defaults to favoring intra-job concurrency when the load is low.

The other two mechanisms of ShuffleWatcher exploit the fact that in multi-tenancy, each job inevitably experiences reduced concurrency due to resource sharing with other jobs. ShuffleWatcher employs *Shuffle-aware Map Placement (SAMP)* on the Map side to trade this reduced concurrency with higher Shuffle locality. SAMP is based on the following assertion. Given a favorable Shuffle and Reduce schedule, a job's Map assignment (e.g., to as few sub-clusters as possible) that optimizes Map plus Shuffle locality results in higher network traffic reductions compared to one that optimizes Map locality alone as done by previous schedulers. SAMP leverages input data replication to optimize the sum of Map and Shuffle locality. In contrast to previous schedulers, SAMP may forgo some Map locality to achieve higher Shuffle locality. The favorable Shuffle and Reduce schedule is ensured by the other two mechanisms of ShuffleWatcher.

ShuffleWatcher employs *Shuffle-aware Reduce placement (SARP)* on the Reduce side to achieve higher Shuffle locality. Previous schedulers assign Reduce tasks to whichever node becomes free, assuming a uniform Map placement, and consequently, Map output distribution throughout the cluster. Such an assumption that may generally hold true for single-tenancy is no longer valid for multi-tenancy due to reduced concurrency per job. As a result, previous multi-tenant schedulers unnecessarily spread out a job's Shuffle in the cluster. SARP is based on the following assertion. Given a favorable Map and Shuffle schedule, a job's Reduce assignment that optimizes Shuffle locality results in higher network traffic reductions compared to one that randomly distributes Reduce tasks. The favorable Map and Shuffle schedule is ensured by the other two mechanisms of ShuffleWatcher. In ShuffleWatcher, most of the Map tasks finish before the Reduce tasks are scheduled whenever NASS delays the Shuffle and, therefore, the distribution of the intermediate data is known. SARP preferentially assigns each job's Reduce tasks to sub-clusters based on how much of the job's intermediate data they contain. Thus, SARP localizes most of the Shuffle and reduces cross-bisection Shuffle traffic.

We implement ShuffleWatcher in Hadoop [21] combined with Fair Scheduler [31]. On a 100-node Amazon EC2 cluster, ShuffleWatcher achieves 46% higher throughput and 48% reduced network traffic compared to Delay Scheduling [39] while improving job turn-around times by 32%. One may think that by trading-off intra-job concurrency for Shuffle locality, ShuffleWatcher may sacrifice turn-around times to gain throughput; on the contrary, by improving Shuffle locality and temporally balancing Shuffle traffic, ShuffleWatcher improves turn-around times, not only on average but of all 300 jobs in our experiments.

The rest of the paper is organized as follows. We provide a brief overview of multi-tenant scheduling in Section 2, and describe ShuffleWatcher in Section 3. We present our experimental methodology in Section 4, and results in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Background, Challenges and Opportunities

We provide a brief background on scheduling in MapReduce clusters and discuss challenges and opportunities offered by multi-tenant clusters.

2.1 MapReduce Job Execution

We begin with the aspects of a MapReduce job's execution that are relevant to multi-tenant scheduling.

When executing a MapReduce job, Map and Reduce tasks are scheduled to maximize concurrency (i.e., occupy the entire cluster or as much as possible). Consequently, the all-Map-to-all-Reduce Shuffle results in an all-nodes-to-all-nodes communication, which stresses the network bisection bandwidth [13,17,35,37,39].

To improve performance, a job's Shuffle is overlapped with its own Map phase (i.e., the Shuffle of data produced by earlier Map tasks occurs while later Map tasks execute). To achieve this overlap, the scheduler must assign Reduce tasks (which perform the Shuffle) to nodes very early in the Map phase, before most of the Map tasks have even begun execution. Two key implications of this approach are: (1) The Shuffle's schedule is fixed rigidly relative to the Map phase and cannot change dynamically in response to network load, and (2) the distribution of intermediate data is not known when the Reduce tasks are assigned. In single-tenancy, because the Map tasks are spread out across the entire cluster, a random assignment of Reduce tasks to nodes is close to optimal because it not only exploits full concurrency. but also achieves the overlap with the Map phase.

To reduce network traffic by exploiting locality, the scheduler attempts to assign a Map task to either the node or the rack that holds the task's input data. However, Shuffle locality is not considered because Reduce tasks are scheduled well before the distribution of Map output is known to gain the above-mentioned benefits.

2.2 Multi-tenant Scheduling

In a multi-tenant environment, users submit jobs to the scheduler, which enqueues and assigns the jobs' Map and Reduce tasks to nodes based on a specified fairness policy. Many fairness policies have been proposed. For example, Fair Scheduler [31] uses a share-based order among users, while Dynamic Resource Fairness (DRF) [16] ensures that the critical resource shares are equalized across users. From a scheduling perspective, the fairness policy effectively determines to which user's jobs should an available node be allocated. Among the chosen user's tasks, the earliest enqueued task that fits resources of the available node is scheduled. Under multi-tenancy, a single job does not have access to the full resources of the cluster, and therefore sees an inevitable reduction in concurrency.

Previous multi-tenant schedulers preserve Map-input locality by scheduling Map tasks on nodes or racks that have the corresponding input data. They also retain the approach of overlapping Map and Shuffle phases within each job, spread out the Map and Reduce tasks of a job for maximal concurrency, and assign the tasks to nodes irrespective of how much Shuffle data they consume.

In this context, we wish to improve throughput and job latency while obeying the specified fairness criteria.

2.3 Challenges and Opportunities in Multi-tenant Scheduling

The key challenge in multi-tenancy is that the execution of multiple concurrent shuffle-heavy jobs severely stresses the network bisection bandwidth. Previous schedulers optimize for Map-input traffic but not for Shuffle traffic. Moreover, they do not differentiate between Shuffle-heavy and Shuffle-light jobs and may concurrently schedule multiple Shuffle-heavy jobs, worsening the impact of network saturation. Such saturation affects all running jobs (not just the Shuffle-heavy jobs), and severely degrades cluster throughput as well as individual job turn-around times.

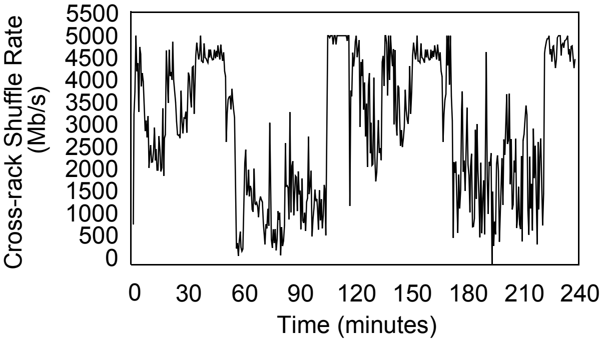


FIGURE 1: Shuffle profile in 100-node EC2 multi-tenant cluster

While network switch and link bandwidth scale with hardware technology, bisection bandwidth is a global resource that is expensive to scale up with the cluster's computational resources. Previous work [4] has proposed new topologies that achieve high bisection bandwidth without requiring custom, high-end switches. Nevertheless, provisioning for peak network bisection bandwidth requirements is still quite expensive, and wasteful because the full bandwidth is not utilized at all times [42]. Hence, clusters typically provide lower bandwidths at the aggregation and core layers of the network topology than at the cluster edges (i.e., the links to nodes). This bandwidth over-subscription results in significant cost savings. Large clusters usually have bandwidth over-subscription ratios ranging from 5:1 to 20:1 or even higher. Therefore, when all nodes are concurrently communicating (as in Shuffle), the bisection bandwidth available *per node* is still much less than bandwidth available within a rack (e.g., 50-200 Mbps compared to 1 Gbps within rack [13,17,35,37,39]).

While multi-tenancy poses the above challenge, it also offers new opportunities.

- Multi-tenant workloads often include a significant fraction of shuffle-light jobs [9,39], which may be overlapped with shuffle-heavy jobs without exacerbating the load on the cluster network. Current schedulers are Shuffle-unaware, resulting in periods of relatively high and low Shuffle activity in the cluster. Figure 1 shows the measured Shuffle traffic vs. time in a 100-node Amazon EC2 cluster running a workload mix representative of Yahoo and Facebook clusters [9,39]. From the figure, we see that network load is quite bursty and saturates the network during some periods, while leaving it under-utilized during other periods. This creates an opportunity to create a more temporally balanced network load.
- Unlike single-tenancy, where intra-job Map-Shuffle overlap is critical and delaying the Shuffle invariably hurts performance, multi-tenancy affords the possibility of achieving such overlap across jobs, creating an opportunity to flexibly schedule a job's Shuffle.
- In multi-tenancy, each job gets only a fraction of the cluster resources for its execution. Such reduced concurrency results into a skewed intermediate data distribu-

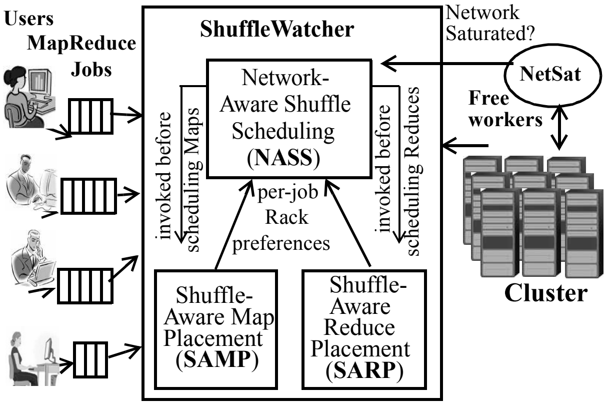


FIGURE 2: Overview of ShuffleWatcher

tion, creating an opportunity to exploit Shuffle locality.

ShuffleWatcher exploits these opportunities to shape and to reduce the Shuffle traffic as described in the next section. Note that delaying Shuffle, or localizing Map and Reduce tasks of a job can be achieved without losing fairness by exploiting the choice among a user's many jobs and tasks. For example, a user whose Reduce task is delayed to alleviate network load need not lose her turn. Instead, ShuffleWatcher schedules a Map or Reduce task of the same user whose input or intermediate data is present on the node. In effect, ShuffleWatcher operates within the confines of a specified fairness policy.

Many of the opportunities described above require curbing a single job's concurrency. We show that such curbing can be done without hurting (and on the contrary, often improving) both cluster throughput and job turn-around times. Multi-tenancy implies that the cluster is shared among multiple jobs, so the concurrency available to each job is anyhow restricted. The aforementioned choice among a user's jobs and tasks is typically sufficient to fully utilize cluster's resources, and any loss in concurrency for a job is more than offset by the significant performance improvement due to Shuffle locality.

3 ShuffleWatcher

Figure 2 shows a high-level overview of ShuffleWatcher. Like other multi-tenant schedulers, ShuffleWatcher receives job submissions from one or more users. The scheduler monitors the status of nodes in the cluster, and schedules Map and Reduce tasks to them as they become available. ShuffleWatcher consists of three components: *Network-Aware Shuffle Scheduling (NASS)* (Section 3.1), *Shuffle-Aware Map Placement (SAMP* (Section 3.2), and *Shuffle-Aware Reduce Placement (SARP)* (Section 3.3).

3.1 Network-aware Shuffle Scheduling (NASS)

Invoked when a worker on rack *r* requests a task

```
1.  Select user based on fairness policy.
2.  if (NetworkSaturated) {
3.    find a task of selected user in the following order:
4.        Map task for which r is in PreferredMapRacks (from SAMP)
5.        Local Map task of any job
6.        Any available Map task
7.        Reduce task of any Shuffle-light job
8.        Reduce task of Shuffle-heavy job for which
              PreferredReducesPerRack[r] is not met (from SARP)
9.        Any available Reduce task
10. }
11. else {
12.   find a task of selected user in the following order:
13.       Reduce task of Shuffle-heavy job for which
              PreferredReducesPerRack[r] is not met (from SARP)
14.       Reduce task of any Shuffle-heavy job
15.       Any available Reduce task
16.       Map task for which r is in PreferredMapRacks (from SAMP)
17.       Local Map task of any job
18.       Any available Map task
19. }
```

FIGURE 3: NASS Algorithm

Figure 3 shows a high-level overview of steps performed by NASS, which is invoked whenever a worker node in the cluster requests a new task. First, NASS picks a user to whom the node should be allocated as per fairness criteria (line 1) which can be based on any of the policies proposed previously [16,21,30,31]. Tasks only from the selected user's jobs are considered in the remaining steps (lines 2-19) of NASS to ensure the user's fair share.

The remaining steps in NASS, which differ significantly from previous MapReduce schedulers, are responsible for shaping the Shuffle traffic by exploiting the concurrency-locality trade-off (Section 2). This trade-off is driven by the network load as monitored by a daemon, called *NetSat*, which periodically determines each node's cross-rack traffic of all jobs due to the Shuffle, remote Map input reads, and Reduce output writes. NetSat compares the ratio of the traffic and the cross-rack bandwidth available to the node against a threshold, called *NWSaturationThreshold*, to set a flag, called *NetworkSaturated,* when the ratio exceeds the threshold. We found that *NWSaturationThreshold* can be in the broad 75-100% range and result in less than 1% difference in cluster throughput. While our current NetSat implementation uses only the limited notions of within-rack and cross-rack traffic, more precise information about the network topology or network congestion monitoring mechanisms, when available, can be used. Similarly, while *NetSat* currently monitors network traffic only due to MapReduce jobs in the cluster, the daemon can be modified to account for traffic from other applications running concurrently in the cluster (e.g., interactive workloads and MPI jobs).

If *NetworkSaturated* is true, NASS orders the Map and Reduce tasks so as to reduce the load on the network (lines 3-9). In this ordering, NASS curbs intra-job Map-Shuffle concurrency by preferring Map tasks and delay-

Invoked when a new job *j* is submitted

```
1. Sort racks in decreasing order of input data for j
2. TmpMapRacks = {}
3. CrossRackTraffic = infinity
4. do {
5.   remove first rack r in sorted list and add to TmpMapRacks
6.   estimate CrossRackTraffic = remote Map traffic + cross-rack
          Shuffle //assumes SARP is used.
7. } while (CrossRackTraffic decreases)
8. PreferredMapRacks = TmpMapRacks
9. compute TentativeReducesPerRack for SARP assuming Map
          tasks are scheduled on PreferredMapRacks
```

FIGURE 4: SAMP Algorithm

ing Reduce tasks and the associated Shuffle. NASS looks for a Map task of the selected user in the following categories listed in the order of increasing network load (lines 4-6): Map tasks for which the rack of the node requesting work is in SAMP's *PreferredMapRacks* (line 4) as described in Section 3.2; remaining local Map tasks (line 5), and remaining Map tasks (line 6). For local Map tasks, NASS obeys the locality-driven Map scheduling typically used in previous schedulers where node-local and rack-local tasks are explored in that order. In some cases, SAMP may explicitly decide that incurring some remote Map tasks reduces the total (remote Map + Shuffle) traffic. Such Map tasks are covered in SAMP's *PreferredMapRacks***.** Within each category, the tasks are ordered by job arrival times.

If there is no available Map task, NASS looks for a Reduce task of the selected user in the order of increasing Shuffle volume (lines 7-9). The jobs are categorized into *Shuffle-heavy* and *Shuffle-light* based on the Shuffle-to-Map-input volume ratio, called *ShuffleInputRatio (*ratio > 1.0 indicates a Shuffle-heavy job). Our current implementation initializes this ratio to be 1.0 and dynamically updates the value as the Map phase of a job progresses. This ratio could also be provided by the user, if known in advance, or tracked from previous runs of the job. NASS's preferred order for Reduce tasks is: Shuffle-light (line 7), Shuffle-heavy from a job for which fewer than the desired number of Reduces as identified by SARP's *PreferredReducesPerRack* have been executed (line 8) as described in Section 3.3, followed by any Reduce task (line 9).

If *NetworkSaturated* is false, NASS defaults to favoring high intra-job concurrency by prioritizing Reduce tasks (and hence the Shuffle) over Map tasks (lines 12-18). Accordingly, NASS prioritizes Shuffle-heavy Reduce tasks preferred by SARP, followed by Reduce tasks of any Shuffle-heavy job to fully utilize the available bandwidth followed by Reduce tasks of any Shuffle-light job (lines 13-15). In the absence of a Reduce task, NASS schedules a Map task following the same preference order as in the saturated-network case to

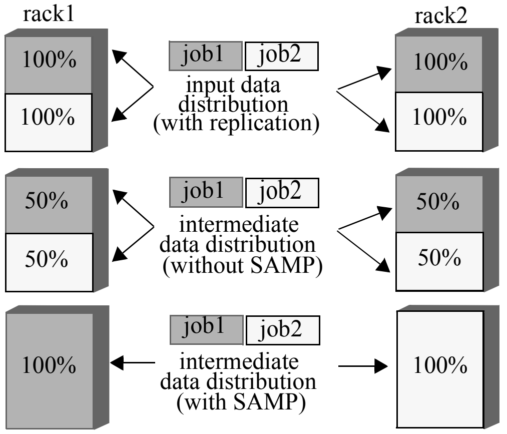


FIGURE 5: Cross-rack Shuffle Reduction with SAMP

improve locality (lines 16-18). Because NASS is guaranteed to choose either a Map task (line 6 and line 18) or a Reduce task (line 9 and line 15) of the selected user irrespective of network saturation, NASS maintains fairness. One may think that because NASS maintains per-user fairness but not per-job fairness, NASS may either hurt the turn-around times of some jobs or not perform well under per-job fairness. We address both these concerns in our results and show that neither of these concerns is true (Section 5.2 and Section 5.4, respectively).

3.2 Shuffle-aware Map Placement (SAMP)

Recall from Section 2.3 that in multi-tenancy, each job experiences reduced concurrency resulting into a skewed intermediate data distribution. The Map-input locality driven Map scheduling employed in previous MapReduce schedulers [21, 39] does not consider Shuffle locality. SAMP goes beyond previous schedulers in two ways: (i) it leverages input data replication to localize intermediate data for a job, by prioritizing the execution of some replicas over others, and (ii) it allows remote execution of Map tasks when the resulting remote Map input traffic is outweighed by the Shuffle traffic reduction due to localized intermediate data. Such restriction of a job's Map tasks to a subset of nodes or racks achieves high Shuffle locality at the expense of full intra-job concurrency, which is anyway not available in multi-tenancy. SAMP relies on NASS and SARP to exploit Shuffle locality in later phases of a job execution.

The procedure used by SAMP is shown in Figure 4. SAMP is triggered once per job, at the time of job submission. Based on the locations of a job's input data, SAMP prepares a sorted list of racks in decreasing order of the amount of the job's input data that they contain (line 1). SAMP initializes a list of racks, *TmpMapRacks*, (line 2) and a variable *CrossRackTraffic* to measure cross-rack traffic (line 3). Next, SAMP keeps adding racks to *TmpMapRacks* in the sorted order, and computes the resulting *CrossRackTraffic* as the sum of remote Map traffic incurred and cross-rack Shuffle vol-

Invoked when job *j* schedules its Reduce tasks

```
1.if (fraction of Maps completed > MapCompletionThreshold)
2.    for each rack r
3.      PreferredReducesPerRack[r] = NumReduces * Intermediate
              data size on rack r / Current Intermediate data size of j
4.} else
5.    for each rack r
6.      PreferredReducesPerRack[r] = TentativeReducesPerRack[r]
```

FIGURE 6: SARP Algorithm

ume until this sum is minimized (lines 4-7). Remote Map traffic is estimated from the fraction of the job's input data that does not reside on the racks in *TmpMapRacks*, whereas cross-rack Shuffle volume is estimated based on the job's *ShuffleInputRatio* (Section 3.1) and assuming SARP's Reduce placement. The final list of racks *TmpMapRacks* is assigned to *PreferredMapRacks* (line 8) which is then communicated to NASS for task scheduling (Section 3.1). SAMP also computes an estimated number of Reduce tasks for each rack (*TentativeReducesPerRack)* assuming that the Map tasks are scheduled on *PreferredMapRacks*. *TentativeReducesPerRack* is used by SARP when SARP is invoked after only a few Map tasks complete, (i.e., intermediate data locations are unavailable) (Figure 6, lines 4-6).

To highlight the advantages of SAMP, Figure 5 shows a simple example with two jobs, *job1* and *job2*, whose input data is replicated and available on two racks, *rack1* and *rack2*. Because the nodes of each rack become available to execute tasks at roughly the same rate, previous schedulers would assign equal numbers of Map tasks for each job to the nodes within each rack, without incurring remote Map traffic. However, such scheduling results in uniform intermediate data distribution for both jobs, creating little opportunity for SARP to reduce cross-rack Shuffle traffic. In contrast, SAMP's *PreferredMapRacks* selection (*rack1* for *job1*, *rack2* for *job2*) results in a schedule that places all the intermediate data for *job1* on *rack1* and for *job2* on *rack2*, which SARP can exploit to reduce cross-rack Shuffle traffic for both jobs. In this example, note that SAMP exploits input data replication to avoid remote Map traffic.

3.3 Shuffle-aware Reduce Placement (SARP)

Recall from Section 2.3 that the intermediate data distribution is likely to be skewed in a multi-tenant cluster. The scheduling of Reduce tasks solely based on node availability increases the volume of cross-rack Shuffle traffic. Shuffle-aware Reduce placement (SARP) exploits SAMP's Map assignment and NASS's delayed Reduce scheduling to localize most of the Shuffle within racks. SARP achieves this localization by computing a preferred number of Reduce tasks on each rack based on the amount of intermediate data the rack holds.

SARP's algorithm is shown in Figure 6. SARP is invoked when the Reduce tasks of a job are first enabled for scheduling. This enabling is done when either the Map phase is complete or there are unoccupied slots for the user to schedule a task. SARP first checks whether sufficient number of Map tasks have been completed for the job and significant intermediate data has been accumulated, in order to decrease the chances of poor rack preferences. This check is done by comparing the fraction of completed Maps to a threshold, *MapCompletionThreshold* (line 1). Because NASS schedules Reduce tasks as late as possible, this criterion is satisfied in the common case. SARP then computes (lines 2-3) the preferred number of Reduce tasks for the job on a rack, *PreferredReducesPerRack*, by multiplying the job's total number of Reduce tasks (typically specified by the user in current systems) with the fraction of intermediate data residing on the rack. We found that *MapCompletionThreshold* can be in the broad 5-25% range for less than 1% difference in cluster throughput.

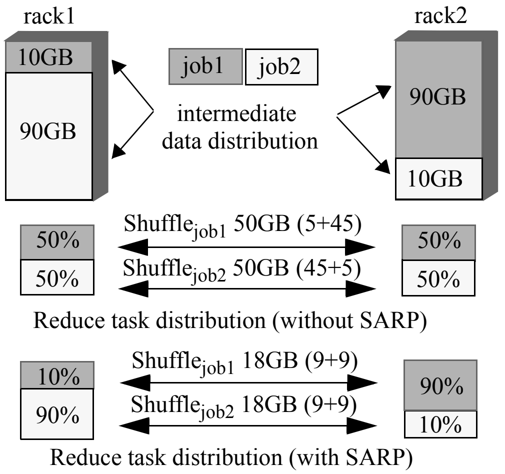

rack1
rack2
10GB
90GB
job1
job2
intermediate data distribution
90GB
10GB
50%
50%
Shuffle$_{job1}$ 50GB (5+45)
Shuffle$_{job2}$ 50GB (45+5)
50%
50%
Reduce task distribution (without SARP)
10%
90%
Shuffle$_{job1}$ 18GB (9+9)
Shuffle$_{job2}$ 18GB (9+9)
90%
10%
Reduce task distribution (with SARP)

FIGURE 7: Cross-rack Shuffle reduction with SARP

In the (uncommon) case when SARP is invoked for a job before sufficient Maps have been executed, SARP relies on SAMP's predictive analysis to decide how many Reduce tasks should be executed on each rack (*TentativeReducePerRack* (Section 3.2)) (lines 5-6).

SARP computes the preferred number of Reduce tasks per rack but does not specify which Reduce task is scheduled on which rack. Because the intermediate data on each rack is likely to contain most or all of the keys (of MapReduce key-value pairs), any Reduce task can be scheduled on a given rack. Therefore, NASS chooses as many available Reduce tasks as specified by SARP.

One may think that SARP's localization may unbalance load across nodes or racks. Because a free node is always assigned some work (preferred or otherwise -line 9 or 18 in Figure 3), such imbalance does not occur.

Figure 7 illustrates a simple example showing SARP's Shuffle traffic reduction in a cluster for two jobs whose data resides only on two racks. When the Reduce

Table 1: Benchmarks Characterization

Shuffle-heavy	terasort(5%), ranked-inverted-index(10%), self-join(10%), word-sequence-count(10%), adjacency-list(5%)
Shuffle-medium	inverted-index(10%), term-vector(10%)
Shuffle-light	grep(15%), wordcount(10%), classification(5%), histogram-movies(5%), histogram-ratings(5%)

tasks are evenly distributed among the racks, both jobs need to transfer half (50GB) of their intermediate data from one rack to the other. With SARP, each job needs to transfer only 18GB of intermediate data across the racks reducing the cross-rack Shuffle traffic by 64%. The reductions in Shuffle traffic is even greater for cases where previous schedulers' random placement assigns all Reduce tasks of job1 to rack1 and of job2 to rack2.

3.4 Discussion

We end our description of ShuffleWatcher by providing a few additional insights and clarifications.

One possible alternative to ShuffleWatcher is simply to reduce the Shuffle traffic by assigning Reduce tasks to machines or racks in proportion to the distribution of the intermediate (Map output) data or the input data. While ShuffleWatcher achieves the same effect by preferentially assigning Reduce tasks to the nodes containing the intermediate data, the alternative approach is simpler. However, such an approach does not consider other equally-important aspects of multi-tenancy, such as job latency, cluster utilization, and fairness. For example, to schedule Reduce tasks based solely on the intermediate data distribution, the scheduler must delay the Shuffle until the Map phase is complete, entirely losing the opportunity for intra-job Shuffle-Map concurrency and potentially increasing job latency. Alternately, scheduling Reduce tasks based solely on the input data distribution eliminates exploiting any additional skew in the intermediate data. In both cases, fixing the assignment of Reduce tasks to machines leaves the scheduler with limited flexibility. If none of a job's tasks can be executed on a free machine due to the assignment, then resources are under-utilized. As such, the scheduler must reduce Shuffle volume while considering these other important aspects, which preclude a simple or fixed assignment and necessitate the more complete approach of ShuffleWatcher. Finally, the alternative approach does not temporally shape the Shuffle traffic and therefore does not capture a significant part of ShuffleWatcher's improvements (more than 40% in Figure 12).

The mechanisms for tracking job execution, fault tolerance, straggler identification and backup task execution, are not modified by ShuffleWatcher.

In rare cases, Shuffle-aware scheduling employed by ShuffleWatcher may result in a particular job getting starved due to unavailability of preferred racks (caused by load or failure), while other jobs from the same user are executed to satisfy the fairness constraint. ShuffleWatcher addresses this problem by tracking job submission times at the granularity of a window such that a user's jobs submitted in an earlier window are prioritized over those submitted in a later one, overriding the heuristics in NASS, SAMP and SARP. The window width acts like a time-out interval and can be set as some multiple of the average job completion time. A window of 10 minutes was enough to prevent starvation of any jobs in our cluster.

Table 2: Distribution of job sizes

Input job sizes	% jobs	Input job sizes	% jobs
< 100MB	20%	100GB - 200GB	10%
100MB- 1GB	19%	200GB - 500GB	7%
1GB - 20GB	21%	500GB - 1TB	8%
20GB - 100GB	10%	> 1TB	5%

By default, ShuffleWatcher improves performance while strictly obeying the constraints provided by any fairness policy (we evaluate ShuffleWatcher using two such policies in Section 5). However, ShuffleWatcher can be operated under relaxed fairness constraints (e.g., as employed in Delay Scheduling [39]). We evaluate the impact of relaxed fairness constraints in Section 5.

Finally, although ShuffleWatcher performs additional steps (Figure 2) compared to current schedulers, these steps do not impact scalability as they are executed either periodically at a low frequency (NetSat) or only once per job (SAMP and SARP). The computation in NASS is quite simple, and adds negligible overheads.

4 Experimental Methodology

We implement ShuffleWatcher in Hadoop (version 1.0.0) [21], and evaluate on a 100-node testbed in Amazon's Elastic Compute Cloud (EC2) [5].

4.1 Testbed

In the 100-node cluster, we use "extra-large" instances, each with 4 virtual cores and 15 GB of memory. EC2 does not provide any information about the underlying network topology or physical locations of the instances. In large clusters, the cross-rack bandwidth is usually much lower than the within-rack bandwidth [13, 21, 25, 39]. To emulate a cluster with realistic bandwidths and to distinguish the nodes from each other based on their location (e.g., rack-local versus rack-remote), we divide our cluster into 10 sub-clusters of 10 nodes each. We identify the sub-clusters by their elastic IP addresses assigned based on their location in EC2. We use the network utility tools *tc* and *iptables* to limit the aggregate bandwidth from one sub-cluster to another to 500 Mbps (50 Mbps is the typical per-node bisection bandwidth [13, 37, 35, 12]), without limiting the bandwidths within each sub-cluster. Because the aggregate bandwidth, and not individual link bandwidths, is limited, a subset of nodes can fully utilize the entire aggregate bandwidth when the other nodes are not using their links. We measure the bandwidth within each sub-cluster to be around 250 Mbps, resulting in the ratio of cross-rack and within-rack per-node bandwidths to be 5:1 at peak network utilization. This ratio being at the lower end of typical over-subscription ranging from 5:1 to 20:1 or even higher makes our results conservative; a higher ratio would mean lower bisection bandwidth making ShuffleWatcher even more important. Similarly, using a shared cluster such as EC2 instead of a dedicated cluster makes our results conservative and realistic because the shared cluster comes with network traffic interference from jobs outside ShuffleWatcher's control which would be the case in real deployments. This interference impacts the accuracy of *NetSat's* estimate of network saturation, despite which ShuffleWatcher achieves significant improvements.

4.2 Multi-tenant Scheduler Implementations

We implement ShuffleWatcher on top of two fairness schemes, Fair Scheduler [31] and Dominant Resource Fairness (DRF) [16]. We compare ShuffleWatcher to these baselines as well as Delay Scheduler [39]. Delay Scheduler implementation is open-source and is available with Hadoop release. Delay Scheduler is implemented on top of Fair Scheduler and exploits relaxed fairness among users. For a fair comparison with Delay Scheduler, we configure ShuffleWatcher with Delay Scheduler's relaxed fairness constraints. For the Fair Scheduler-based implementations, each node concurrently runs four Map tasks and two Reduce tasks. DRF is another scheduler based on generalized min-max fairness algorithm [16]. Because DRF's implementation is not publicly available, we develop one. To determine a job's CPU and memory requirements for DRF, we run each of our benchmark jobs individually and monitor the maximum resources needed, as done in [16].

For ShuffleWatcher, we set *NWSaturationThreshold*, the per-rack link utilization threshold to measure network saturation (Section 3.1), to 400 Mbps which is 80% of the admissible bandwidth capacity. We set *MapCompletionThreshold,* the fraction of Map tasks to be completed after which SARP computes the preferred locations for Reduce tasks based on actual intermediate data rather than on SAMP's prediction (Section 3.3), to be 15%. Because NASS considers SAMP's preferences during scheduling, the actual intermediate data and SAMP's predictions are so close that our performance improvements are not sensitive to variations in this parameter. We choose the default distributed file system (HDFS) block size of 64 MB and replication factor of 3.

Table 3: Traffic Volume (GB) under Fair Scheduler

Job Type	Total Shuffle	Cross-Rack Shuffle	Remote Map Traffic	Total Cross-Rack Traffic
Shuffle-heavy	1261	1108	187	1295
Shuffle-medium	70	63	38	101
Shuffle-light	13	11	48	59

4.3 Workloads

We use typical workloads consisting of benchmarks drawn from the Hadoop release and [2]. Based on *ShuffleInputRatio* (Section 3.1), we characterize the benchmarks as Shuffle-heavy, Shuffle-medium or Shuffle-light in Table 1. The table also shows the percentage of jobs of each type in the workload. The variation in job mixes and the variation in job input sizes (Table 2) are based on real workloads from Yahoo and Facebook [9].

We set the number of users to 30 for our 100-node cluster, consistent with the Facebook cluster usage reported in [39] (200 users for a 600-node cluster). Job submission follows an exponential distribution [39, 9, 16]. Each user picks a job from our suite in Table 1, although with different input datasets. After testing with different mean job inter-arrival times, we set the mean to be 40 seconds to utilize the cluster maximally for the base case (Delay Scheduler). We use the same job arrival rates for all schedulers.

Because we are interested only in the steady-state period of the cluster under full load, we ignore the load ramp-up and ramp-down periods. We run each experiment for a steady-state duration of 4 hours.

5 Experimental Results

We first show Shuffle's importance (Section 5.1) and then compare ShuffleWatcher against three baselines, namely Fair Scheduler, Delay Scheduler, and DRF Scheduler (Section 5.2). We then isolate the impact of NASS, SARP and SAMP (Section 5.3) Finally, we show the impact of varying the number of jobs per user (Section 5.4) and the job mix (Section 5.5) on ShuffleWatcher's improvements.

5.1 Importance of the Shuffle

Table 3 shows the actual volumes of the total Shuffle (within-rack and cross-rack), cross-rack Shuffle and remote Map traffic for Shuffle-heavy, Shuffle-medium and Shuffle-light jobs under Fair Scheduler. We show cross-rack traffic as a proxy for the bisection bandwidth demand. From the first two columns, we see that most of the Shuffle (~ 90%) is across racks. From the last three columns, we see that the cross-rack Shuffle volume of the Shuffle-heavy jobs has a significant contribution (>75%) of the total cross-rack traffic, and the contribu-

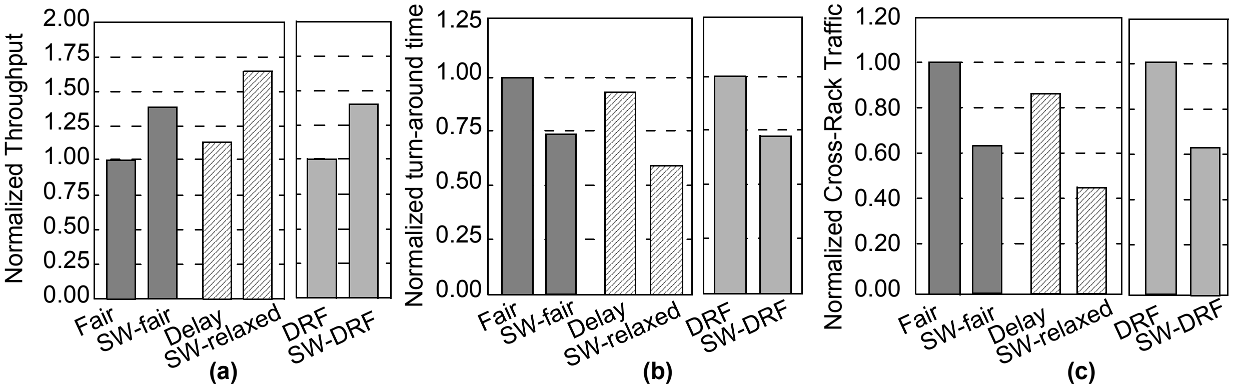


FIGURE 8: Performance comparison

tion of the remote Map traffic is less than 20% of the total cross-rack traffic. These numbers confirm the Shuffle's dominance and justify our focus on the Shuffle.

5.2 Performance

Figure 8 shows ShuffleWatcher operating under two fairness policies — Fair Scheduler (SW-fair) and DRF Scheduler (SW-DRF). To ensure a fair comparison with Delay Scheduler (Delay) which relaxes fairness constraints, we also show ShuffleWatcher configured with similarly relaxed constraints (SW-relaxed). We use relaxed fairness interval of five seconds, consistent with [39]. In the three graphs in Figure 8(a-c) with two subgraphs per graph, the Y-axes represent throughput, turnaround time and cross-rack traffic, respectively. The first sub-graph shows Fair, SW-fair, Delay, and SW-relaxed which use Fair Scheduler's fairness policy and therefore are normalized to Fair. The second sub-graph shows DRF and SW-DRF which use the DRF policy and therefore are normalized to DRF.

ShuffleWatcher (SW-fair) achieves significant improvements over Fair Scheduler, with 39% higher throughput (Figure 8(a)), 27% lower turn-around time (Figure 8(b)) and 36% lower cross-rack traffic (Figure 8(c)). Compared to Delay Scheduler (Delay), ShuffleWatcher (SW-relaxed) achieves more improvements. Specifically, SW-relaxed is 46%, 32%, and 48% better than Delay in throughput, turn-around time, and cross-rack traffic, respectively (computed from the graphs). SW-DRF also achieves similar performance improvements as SW-fair, showing that our improvements are largely independent of the underlying fairness policy. In the rest of the paper, we report results only for SW-fair, because results for SW-DRF are similar.

Figure 9 shows the average intra-job concurrency in Fair Scheduler and ShuffleWatcher measured as the fraction of the allocated per-user slots (resources) occupied by a job's Map and Reduce tasks during first, middle and last thirds of the job's work completion. Because typically Map tasks are numerous and Reduce tasks are fewer, Fair Scheduler's concurrency fraction goes from nearly one for Map phase in the first two-thirds of a job's execution (i.e., one job's Map tasks occupy almost all of the user's slots) to less than half for the Reduce phase in the last third of a job's execution (i.e., one job's Reduce tasks leave vacant slots which are occupied by the user's other jobs). The graph shows that ShuffleWatcher trades off intra-job concurrency for Shuffle locality. While the lower Map and Reduce concurrencies due to SAMP and SARP are obvious, these lower concurrencies also mean lower Map-Shuffle concurrency due to NASS.

Figure 10 shows the measured cross-rack Shuffle traffic over time in our testbed for one of our ShuffleWatcher runs. Comparing this profile with that for Fair Scheduler in Figure 1, we see that the network traffic with ShuffleWatcher is relatively balanced.

To show that ShuffleWatcher does not hurt any job's turn-around time by trading-off intra-job concurrency

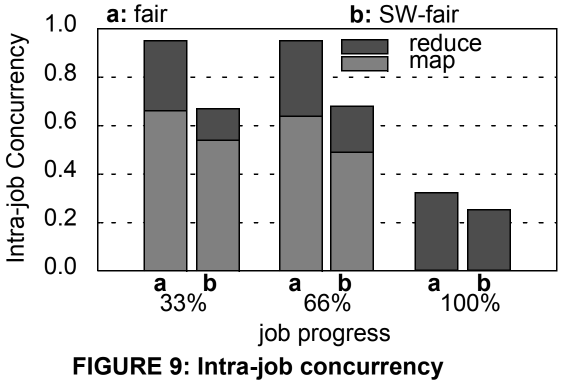


FIGURE 9: Intra-job concurrency

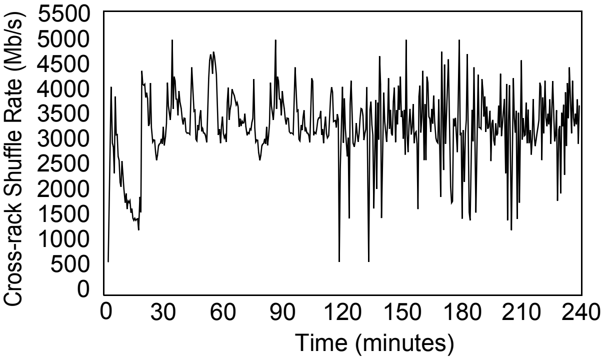


FIGURE 10: ShuffleWatcher: Shuffle profile

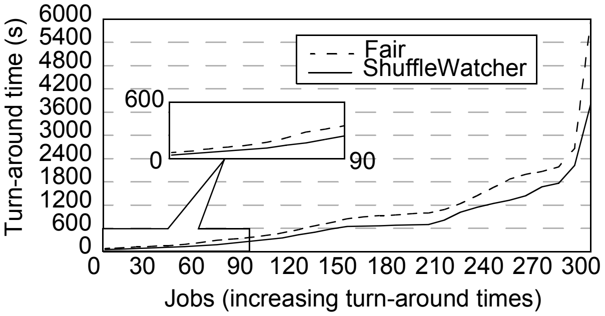


FIGURE 11: Turn-around times of individual jobs

for throughput, Figure 11 plots the turn-around times of individual jobs in Fair Scheduler and ShuffleWatcher. The Y-axis shows the turn-around times and the X-axis shows the jobs ordered in increasing turn-around times under Fair Scheduler (for clarity, jobs 0-90 are shown separately in a blow-up). We see that ShuffleWatcher improves the turn-around time of every one of our 300 jobs, irrespective of the job's size or its Shuffle intensity. This result confirms that by trading-off intra-job concurrency for Shuffle locality in the presence of high contention in multi-tenancy, ShuffleWatcher improves both throughput and turn-around times.

5.3 Impact of NASS, SARP and SAMP

Figure 12 and Figure 13 isolate the impact of NASS, SARP and SAMP by showing a breakdown of ShuffleWatcher's throughput improvements and traffic reduction. Because SARP cannot work without NASS and SAMP cannot work without NASS and SARP, our breakdown is additive in the order of NASS, SARP, and SAMP. We omit the turn-around times breakdown which is similar to the throughput breakdown. In Figure 12 and Figure 13, the Y-axes show throughput and cross-rack traffic, respectively, for ShuffleWatcher normalized to those for Fair Scheduler. We show the breakdown for Shuffle-heavy, Shuffle-medium and Shuffle-light jobs separately, and all the jobs together, to give better insight into ShuffleWatcher's improvements.

From Figure 12, we see that the contribution of each technique is significant across all the three types of jobs. Going from Shuffle-heavy to Shuffle-light, the overall improvement and NASS's contribution increase. Without ShuffleWatcher, the Shuffle-light jobs' short run times are greatly degraded by interference from Shuffle-heavy jobs. ShuffleWatcher, and NASS in particular, reduce this interference, resulting in the observed trend.

The cross-rack traffic breakdown graph in Figure 13 splits the cross-rack traffic into Shuffle traffic and remote Map traffic. From the graph, we see that NASS does not reduce the cross-rack traffic (recall that NASS only shapes, but does not reduce, the traffic). However, SARP, which leverages NASS to improve Reduce-side locality, reduces the cross-rack traffic of Shuffle-heavy, Shuffle-medium and Shuffle-light jobs by 16%, 13% and 4%, respectively, with total traffic reduction of 15%. The reduction due to SARP in Shuffle-light jobs' cross-rack traffic is insignificant because most of the traffic of these jobs is due not to the Shuffle but to remote Map tasks (Table 3) which are not impacted by SARP. Similarly, SAMP, which leverages NASS and SARP to improve Map-side locality, reduces the cross-rack Shuffle traffic by 38%, 23% and 4% for the three job types, while reducing the total traffic by 36%. From the graph, we see that SAMP incurs a small increase in the remote Map traffic for Shuffle-heavy jobs (~3%) to localize the Shuffle to fewer racks, but reduces the total cross-rack traffic volume. Such small increase shows that SAMP successfully exploits data replication to localize the Shuffle without incurring significant remote Map traffic overhead. The graph also shows that the total cross-rack traffic reduction for all the jobs together closely follows that for the Shuffle-heavy jobs which contribute a significant portion of the traffic (Table 3).

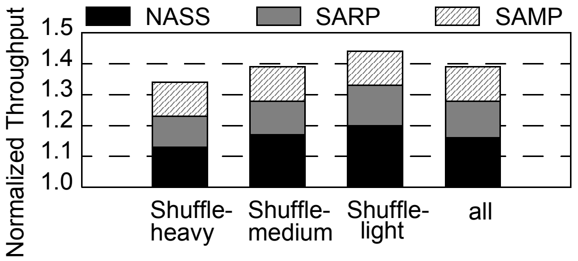


FIGURE 12: Throughput Impact of NASS,SARP,SAMP

5.4 Impact of varying jobs per user

Because ShuffleWatcher exploits the choice among a given user's jobs (i.e., per-user fairness) to adapt to the network load, ShuffleWatcher may not perform well with only one job per user, which is equivalent to per-job fairness. To address this concern, Figure 14(a) shows ShuffleWatcher's sensitivity to the number of jobs per user. We use the same mean job arrival rate as before, but vary the number of jobs per user as 1, 9, 12 (default), and 18. We evaluated the case of one job per user in a local 16-node cluster and the rest of the cases in EC2 as

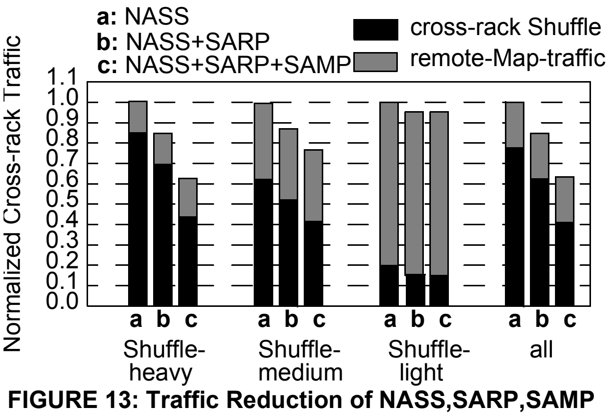


FIGURE 13: Traffic Reduction of NASS,SARP,SAMP

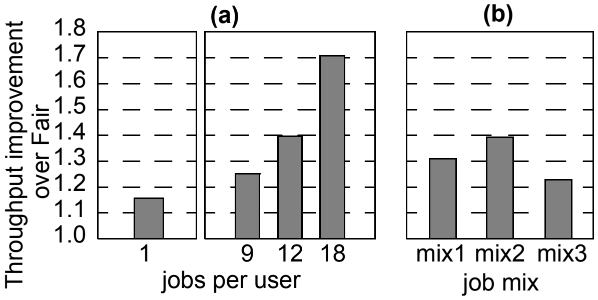


FIGURE 14: Sensitivity to (a) number of jobs per user and (b) job mix

before (we added the first case later, hence the different set up). Therefore, we isolate the one job per user in the left sub-graph while the rest are shown in the right.

The figure shows that even with one job per user ShuffleWatcher achieves 16% higher throughput over Fair Scheduler by choosing between the job's Map and Reduce tasks based on network loading. This result shows that ShuffleWatcher performs well even under per-job fairness. From the right sub-graph, we see that ShuffleWatcher achieves higher improvements with more jobs per user (i.e., under per-user fairness). With more jobs per user, ShuffleWatcher has more choices in its scheduling decisions and, therefore, achieves better network traffic shaping and reduction.

5.5 Sensitivity to the variation in job mix

We show ShuffleWatcher's sensitivity to the job mix in the workload. Figure 14(b) shows throughput improvements over Fair Scheduler for three different mixes of Shuffle-light, Shuffle-medium, and Shuffle-heavy jobs: *mix1* with 20%, 20% and 60%, respectively; *mix2* with 40%, 20%, and 40%, respectively (default); and *mix3* with 60%, 20% and 20%, respectively.

From the figure, we see that ShuffleWatcher improves throughput by 31% and 22% for *mix1* and *mix3*, respectively, compared to 39% for *mix2*. The improvements for *mix1* and *mix3* are significant but lower than that for *mix2* because of reduced opportunity. Compared to *mix2*, *mix1*'s larger fraction of Shuffle-heavy jobs means higher network pressure with fewer low-utilization periods; and *mix3*'s larger fraction of Shuffle-light jobs means lower network saturation. Nevertheless, significant improvements over a wide range of job mixes demonstrate ShuffleWatcher's effectiveness.

5.6 Execution on a dedicated cluster

In addition to 100-node EC2 runs, we also performed runs on a dedicated 16 Xeon-nodes cluster to isolate the interference from jobs outside ShuffleWatcher's control. We scaled down job arrival rate, job sizes, and number of users to match the cluster configuration. We divided the cluster into 4 sub-clusters of 4 nodes each and limited the per-node bisection bandwidth to be the same as in the EC2 cluster (50Mbps). Our results exceeded the performance achieved in the EC2 cluster. ShuffleWatcher achieved 46% higher throughput, 32% lower turn-around time and 48% lower cross-rack traffic over Fair Scheduler. Compared to Delay Scheduler, ShuffleWatcher was 54%, 38%, and 56% better in throughput, turn-around time, and cross-rack traffic, respectively.

6 Related work

Several previous efforts have targeted improving MapReduce performance, including better straggler management [7], improved computation-communication overlap [3,11,36], improved aggregation of intermediate data [43], optimizations for heterogeneous clusters [2,41], and runtime optimizations for iterative MapReductions [8,14,40]. However, these proposals target single-tenancy whereas ShuffleWatcher exploits opportunities that are specific to multi-tenancy.

In the domain of multi-tenancy, Hadoop [21] offers a FIFO scheduler to run jobs in a sequential manner. Capacity Scheduler [30], Fair Scheduler [31] and Dominant Resource Fairness [16] propose different fairness models and schedulers for resource allocation among users. In contrast to their target of achieving fairness, our goal is to improve performance within the given fairness constraints. Delay Scheduling [39] and Quincy [25] target reducing network traffic by optimizing Map-input locality, but not the Shuffle which is by far the most dominant source of traffic in MapReduce. ShuffleWatcher targets the Shuffle by trading-off intra-job concurrency for Shuffle locality to perform better than these previous techniques. Mesos [23] and Yarn [38] facilitate resource provisioning among multiple frameworks that share a cluster (e.g., MPI and MapReduce). These systems decouple resource allocation from job scheduling and can benefit from ShuffleWatcher's scheduling. Purlieus [29] and CAM [26] achieve locality via synergistic placement of virtual machines and input data. However, such static techniques cannot address dynamic variations in the Shuffle traffic.

In the domain of data center networks, researchers have proposed network architectures to improve cluster bisection bandwidth (e.g.,[1,4,15,17,18,19,20,28,33]). Many of these architectures require specialized hardware and/or communication protocols, and thereby incur additional cost especially because bisection bandwidth is inherently hard to scale up. Finally, a few recent papers propose better management of network traffic [10,12,32] without changing the network load, which is often high enough to limit their effectiveness, whereas ShuffleWatcher actively shapes and reduces the network load.

7 Conclusion

We proposed ShuffleWatcher, a Shuffle-aware, multi-tenant scheduler, which counter-intuitively trades-off intra-job concurrency for Shuffle locality. Shuffle-

Watcher employs three mechanisms: *Network-Aware Shuffle Scheduling (NASS), Shuffle-Aware Reduce Placement (SARP),* and *Shuffle-Aware Map Placement (SAMP)* which exploit this trade-off and improve performance by shaping and reducing the Shuffle traffic while working within the specified fairness constraints.

On a 100-node Amazon EC2 cluster running Hadoop, ShuffleWatcher improves cluster throughput by 39-46% and job turn-around time by 27-32% over three state-of-the-art schedulers. Despite trading-off intra-job concurrency for Shuffle locality, ShuffleWatcher does not sacrifice turn-around times to gain throughput; on the contrary, by improving Shuffle locality in the presence of high contention in multi-tenancy, ShuffleWatcher improves turn-around times, not only on average but also of every one of 300 jobs in our experiments. ShuffleWatcher improves both cluster throughput and job latency and, therefore, will be valuable in emerging multi-tenant environments.

References

[1] H. Abu-Libdeh et al. Symbiotic routing in future data centers. In *Proceedings of SIGCOMM,* 2010.

[2] F. Ahmad et al. Tarazu: Optimizing MapReduce on heterogeneous clusters. In *Proceedings of ASPLOS,* 2012.

[3] F. Ahmad et al. MapReduce with Communication Overlap (MaRCO). In *JPDC,* 2012.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of SIGCOMM,* 2008.

[5] Amazon EC2. http://aws.amazon.com/ec2.

[6] Amazon Elastic MapReduce (Amazon EMR). http://aws.amazon.com/elasticmapreduce/.

[7] G. Ananthanarayanan et al. Reining in the outliers in Map-Reduce clusters using Mantri. In *Proceedings of OSDI,* 2010.

[8] Y. Bu et al. The Haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169-190, Apr. 2012.

[9] Y. Chen et al. The case for evaluating MapReduce performance using workload suites. In *Proceedings of MASCOTS,* 2011.

[10] M. Chowdhury et al. Managing data transfers in computer clusters with orchestra. In *Proceedings of SIGCOMM*, pages 98-109, 2011.

[11] T. Condie et al. MapReduce online. In *Proceedings of NSDI,* 2010.

[12] P. Costa et al. Camdoop: exploiting in-network aggregation for big data applications. In *Proceedings of NSDI,* 2012.

[13] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, Jan. 2008.

[14] J. Ekanayake et al. Twister: a runtime for iterative MapReduce. In *Proceedings of HPDC,* 2010.

[15] N. Farrington et al. Helios: a hybrid electrical/optical switch architecture for modular data centers. In *Proceedings of SIGCOMM,* 2010.

[16] A. Ghodsi et al. Dominant Resource Fairness: Fair allocation of multiple resource types. In *Proceedings of NSDI,* 2011.

[17] A. Greenberg et al. VL2: A scalable and flexible data center network. In *Proceedings of SIGCOMM,* 2009.

[18] C. Guo et al. Bcube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of SIGCOMM,* 2009.

[19] C. Guo et al. Dcell: a scalable and fault-tolerant network structure for data centers. In *Proceedings of SIGCOMM,* 2008.

[20] L. Gyarmati and T. A. Trinh. Scafida: a scale-free network inspired data center architecture. *SIGCOMM Comput. Commun. Rev.*, 40(5):4-12.

[21] Hadoop. http://hadoop.apache.org.

[22] Hadoop On Demand. http://hadoop.apache.org/docs/r0.18.3/hod_admin_guide.html.

[23] B. Hindman et al. Mesos: a platform for fine-grained resource sharing in the data center. In *Proceedings of NSDI,* 2011.

[24] M. Isard et al. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of EuroSys,* 2007.

[25] M. Isard et al. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of SOSP,* 2009.

[26] M. Li et al. Cam: a topology aware minimum cost flow based resource manager for MapReduce applications in the cloud. In *Proceedings of HPDC,* 2012.

[27] Torque Resource Manager. http://www.adaptivecomputing.com/products/open-source/torque/.

[28] R. Niranjan Mysore et al. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of SIGCOMM,* 2009.

[29] B. Palanisamy et al. Purlieus: locality-aware resource allocation for MapReduce in a cloud. In *Proceedings of SC,* 2011.

[30] Hadoop Capacity Scheduler. http://hadoop.apache.org/docs/r0.20.2/capacity_scheduler.html.

[31] Hadoop Fair Scheduler. http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html.

[32] A. Shieh et al. Sharing the data center network. In *Proceedings of NSDI,* 2011.

[33] A. Singla et al. Jellyfish: Networking data centers randomly. In *Proceedings of NSDI,* 2012.

[34] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience: Research Articles, UK, Feb. 2005. John Wiley and Sons Ltd.

[35] A. Vahdat et al. Scale-Out Networking in the Data Center. *IEEE Micro*, 30:29-41, July 2010.

[36] A. Verma et al. Breaking the MapReduce Stage Barrier. In *Proceedings of IEEE CLUSTER,* 2010.

[37] D. Weld. Lecture notes on MapReduce(based on Jeff Dean's slides). *http://rakaposhi.eas.asu.edu/cse494/notes/s07-map-reduce.ppt*, 2007.

[38] Yarn. http://hadoop.apache.org/docs/r0.23.0/hadoop-yarn/hadoop-yarn-site/YARN.html.

[39] M. Zaharia et al. Delay Scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of EuroSys,* 2010.

[40] M. Zaharia et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of NSDI,* 2012.

[41] M. Zaharia et al. Improving MapReduce performance in heterogeneous environments. In *Proceedings of OSDI*, 2008.

[42] J. Mudigonda, P. Yalagandula, and J. C. Mogul. Taming the flying cable monster: a topology design and optimization framework for data-center networks. In *Proceedings of USENIX ATC,* 2011.

[43] S. Rao, et.al. Sailfish: a framework for large scale data processing. In *Proceedings of SoCC,* 2012.

Violet: A Storage Stack for IOPS/Capacity Bifurcated Storage Environments

Douglas Santry, Kaladhar Voruganti
NetApp, Inc.

Abstract

In this paper we describe a storage system called *Violet* that efficiently marries fine-grained host side data management with capacity optimized backend disk systems. Currently, for efficiency reasons, real-time analytics applications are forced to map their in-memory graph like data structures on to columnar databases or other intermediate disk friendly data structures when they are persisting these data structures to protect them from node failures. *Violet* provides efficient fine-grained end-to-end data management functionality that obviates the need to perform this intermediate mapping. *Violet* presents the following two key innovations that allow us to efficiently do this mapping between the fine-grained host side data structures and capacity optimized backend disk system: 1) efficient identification of updates on the host that leverages hardware in-memory transaction mechanisms and 2) efficient streaming of fine-grained updates on to a disk using a new data structure called Fibonacci Array.

1. Introduction

Increasingly organizations are finding a lot of business value in performing analytics on the data that is generated by both machines and humans. Not only are more types of data being analyzed by analytics frameworks, but increasingly people are also expecting real-time responses to their analytic queries. Fraud detection systems, enterprise supply chain management systems, mobile location based service systems, and multi-player gaming systems are some applications that want real-time analytics capabilities [10]. In these systems both transaction management and analytics related query processing are performed on the same copy of data. These applications have very large working sets, and they generate millions of transactions per second. In most cases these applications cannot tolerate network and disk latencies, and thus, they are employing main memory architectures [11, 13, 14, 15] on the host application server side to fit the entire working set in memory.

Even though these applications want to store the entire working set in main memory, for protection from node failures, they still need to persist a copy of their data off the application server box. Typically, copies are stored on disk/NAND flash based storage systems because these technologies are much cheaper than main memory. Thus, there is a bifurcation of IOPs optimized data management at the host and capacity optimized data management at the backend disk based storage system.

The key insight that is prompting the work in this paper is that there is a mismatch in the fine-grained data management model on the host and the block optimized data management model in the backend disk/flash based systems. For decades applications and middleware developers have been forced to map their in-memory fine-grained data structures onto intermediate block I/O friendly data structures. Despite the application running entirely in DRAM, the data structures that in-memory databases employ are little changed from when they lived on disks owing to the difficulty in persisting them to a block oriented device. This difficulty is retarding the development of in-memory systems. For ease of implementation the in-memory data structures are part of memory pages that are, in turn, mapped to disk blocks using data structures like B-Trees. However, fundamentally, there are the following inefficiencies in this approach and going forward these have to be resolved in order to provide an efficient end-to-end data management solution for the new emerging real-time analytics applications.

Problem Description

In the past, data structures have been designed to localize updates to a block in order to minimize random I/Os to a disk-based storage system. For example, the inventors of columnar databases observed that if an entire dataset has to be scanned, but only a subset of the columns are important, then a vertical decomposition of a database realized the streaming bandwidth of DRAM and disk subsystems. However, going forward, new types of main-memory graph data structures are emerging, such as *Voronoi Diagrams* [1], that are unconcerned about localizing their updates to a few blocks. These data structures can perform important analytic operations in $O(N)$ time where as a columnar database would require $O(N^2)$ time. *Voronoi Diagrams* are being leveraged in biology, chemistry, finance, archaeology, and business analytics domains. Similarly, middleware software is being designed to support these new emerging graph data structures [10, 17, 20].

When backing up the host side in-memory data structures on to a block-based storage system, it is desirable to be able to efficiently detect and transfer only the updated bytes of data instead of transferring the entire page on which they reside. As is shown in the experiment section of this paper, there are performance benefits in the end-to-end throughput by handling graph data structures natively (the focus of this paper) versus mapping them on to a columnar database or other intermediate data structures [17].

Contributions

In this paper we present a storage system called *Violet* that efficiently marries fine-grained host side data management with back-end block level storage systems. Violet divorces the problem of data structure selection and implementation, which should be wholly based on asymptotic requirements, from data structure persistence, which should not be the application writer's problem. The key highlights of this architecture are:

- **End-End Data Management Stack:** *Violet* presents a byte-oriented storage system that provides an integrated end-to-end storage stack that 1) efficiently detects fine-grained updates on the host 2) replicates the updates remotely on to a back end block based storage system and 3) efficiently streams the updates to a disk drive. The violet storage system architecture has both a host-side footprint and a backend capacity layer footprint.
- **Efficient identification of fine-grained updates:** Violet leverages a hardware transactional memory CPU instruction set (e.g. TSX instruction set from Intel) to detect the read/write changes to data within an in-memory transaction boundary. This allows Violet to track changes at a very fine-grained level in a multi-core CPU environment, and in turn, transfer changes (not the entire data block) off the node in an efficient manner. Violet also allows for both fine-grained data structure level snapshots at the host and coarser grained file level snapshots at the backend capacity optimized storage system.
- **Efficient streaming of sparse random I/Os on to disks:** Violet proposes a new data structure called the *Fibonacci Array* that enhances standard log structured merge trees [12] and COLA [2] data structure notions by leveraging the key insight that in the emerging main memory middleware/application architectures, the backend disk based systems primarily handle write operations as reads are mostly satisfied at the host servers.

The net result of the above innovations is that we are able to efficiently 1) detect 2) transmit and 3) persist fine-grained updates at the host to a block based backend storage system. Thus, emerging real-time applications and databases that are providing support for graph like data structures can leverage the benefits of the techniques being presented in this paper when designing their logging and off-node replication mechanisms.

2. Architecture

As shown in Figure 1, Violet is a distributed system that works as a cluster of cooperating machines. Violet is comprised of 1) a user-level library that gets linked with applications, 2) Violet servers that run on machines with disks attached (called *Sponges*) and 3) a master server that is in charge of the cluster. Applications manage memory with the user space library. The memory region is called a file; when storage class memory (SCM) becomes available Violet can mmap(2) a SCM file and the applications remain unchanged.

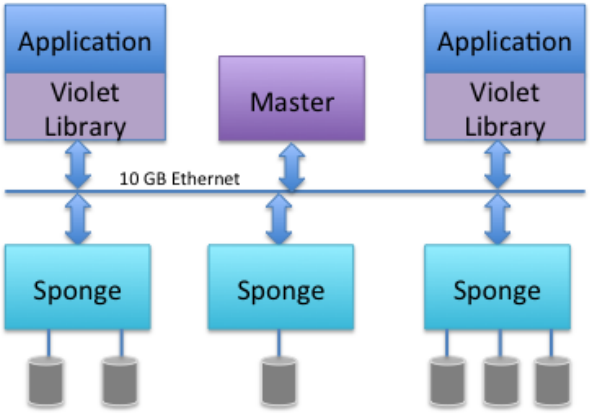


Figure 1: *The Violet Architecture*.

The era of SCMs has started with many vendors announcing the availability of different types of SCMs like PCM, ReRAM, and STT-MRAM [5].

The *primary* copy of a file is on the host in DRAM where the application runs. The copy on the host is a region of volatile anonymous pages created with mmap; so it is not a file on the host per se but this will change with the availability of SCM.

Sponges store the data on their locally attached disks. The set of Sponges that store a file on disk is called the file's replication group. The copies are used to provide disaster recovery and data management. Note that the Sponges only contain a *redundant* copy of a file on their disks. Violet is *not* a distributed memory system. The only updates to the file take place in the application's address space on the host machine. A file can only be written by a single application (the single-writer model is used by most commercial in-memory DBs).

Disks are much cheaper than DRAM, and are expected to be much cheaper than SCM, so basing node failure recovery on disks makes economic sense. As will be seen, a disk based failure recovery strategy does increase restore time (and hence downtime in the event of failure). An alternative approach would be to simply mirror in DRAM on a second host. This would increase the cost of the system accordingly, but it would increase availability as there would be a `hot' host to restart an application on in the event that its machine fails. The system presented here can support both the model where the second copy is on a peer, with the recovery copy on the backend disk, and also the model where the second copy is in DRAM.

3. Host-Side Client Library

This section describes the details of the host-side user space library. We present a C++ API that applications use to describe transactions, and the Violet library that must be linked. The Violet transaction machinery and how it replicates the memory updates to remote storage controllers is described.

The User-Land Storage Stack

The goal of Violet software is to provide disaster recovery and transaction support for applications that desire byte-oriented data structures. The most efficient place to do this is in user-land inside the application. While it is common to persist updates to a mmap'ed file with a page daemon running in the kernel, this would be too inefficient for the application update behavior envisaged. The updates observable by the kernel are at the granularity of a page. Page granularity is too coarse (e.g. the update may be a 4 byte pointer in a linked list node). To efficiently persist these kinds of transactions, the granularity should be as fine as a byte. The desired granularity could be obtained by adding system calls that support the specification of a transaction, but this would impose significant over-head, require POSIX approval and defeats the purpose of mapping the file in the application's address space.

Overview

Violet applications are written in C++ and specify transactions by leveraging an open API. A modified C++ compiler implementing the API compiles the application and instruments the code appropriately to execute the transactions. Finally, the application is linked with the Violet run-time library.

Violet's run-time system is implemented in a user-land library. Applications execute transactions and the library assembles the resulting updates and replicates them to a remote machine.

Transactional Memory

Applications are growing in size and complexity while the number of cores is increasing. As a result, the most common form of thread synchronization, locking, is growing increasingly more onerous. Lock hierarchies must be carefully designed and enforced to avoid deadlock. Selecting an appropriate granularity of locking is crucial to ensure the right balance is struck between parallelism and the cost in both time and space overhead. Priority inversion and lock retention across preemption can be serious artifacts when designing a system. Many data structures, such as balanced binary trees, are notoriously difficult to implement correctly while achieving reasonable performance with locks (e.g. the authors are unaware of any thread-safe red-black tree implementation that did not relax invariants to achieve satisfactory performance).

Herlihy introduced transactional memory [TM] as an alternative to locks to address the above concerns [7]. Transactional memory is a means of safely updating memory in the presence of concurrency that greatly simplifies code when compared with locks. In the last few years there has been strong of revival of interest in transactional memory (e.g. Intel's software TM compiler, GNU libtm). More recently, research database systems such as DBX [6] and HTM [23] have examined the implementation of databases with Intel's hardware transactional memory. Intel, HP, Oracle and others are proposing an update to the C++ language to incorporate TM directly into the language [21]. The proposed support exposes transactional memory as an integral language construct. Ephemeral DRAM data structures are the target use case. The abstraction proposed is of the form: *__transaction {}*. *__transaction* is a new C++ reserved word. The code between the braces would be executed transactionally with ACI (atomicity, consistency and isolation) semantics.

Violet leverages this new C++ reserved word as a means for applications to express relevant memory updates to the system; in effect the C++ proposal is Violet's API. Leveraging C++ increases the likelihood of Violet's adoption since C++ is not proprietary; it is an open standard. Moreover, applications are already being written with the proposed standard. Supporting the API makes the adoption of Violet for applications a *possibility* even as an after thought. Use of a modified compiler is a temporary measure that will be obviated by the implementation of the standard in clang (it is currently only supported in GNU's g++, which has licensing issues).

Violet Transactions

To execute transactions Violet leverages Intel's restricted transactional memory [RTM] feature [18]. RTM is included in Haswell processors' TSX facility. RTM transactions offer ACI semantics, but not durability; TSX is designed to work with ephemeral DRAM. Intel provides two instructions, *xbegin* and *xend*, that demarcate a transaction block. Applications begin a transaction by executing the *xbegin* instruction and commit by executing *xend*. The memory updates following *xbegin* are visible only to the executing thread until the execution of *xend*. Once a transaction commits then all of the updates to memory become visible simultaneously. If a thread writes to a memory location claimed by a peer thread's transaction then the processor aborts the transaction. The reliance on instructions introduced with Haswell TSX means that currently only Haswell TSX processors are supported.

The *xbegin* and *xend* instructions are a means of implementing a transaction, but the Violet library requires further instrumentation in the application. However, as we hope to support stock compilers in the future, reliance on modifications to clang++ must be kept to a minimum. clang++ was modified such that braces in the C++ *__transaction* construct correspond to *xbegin* and *xend* instructions; this is all that can reasonably be expected from a compiler. The remaining instrumentation of the code has to be done externally by a second tool that isolates the proprietary requirements. Violet uses a pre-assembler processor [PAP] to instrument application code. The tool operates on the assembly language emitted by the compiler. PAP scans the assembly code, identifies transactional blocks and then inserts the instrumentation. Transactional blocks are defined as everything in between *xbegin* and *xend* instructions.

The mechanics of a Violet transaction are depicted in Figure 2. The blue boxes are functions inside the Violet library. The arrows represent invocations inserted by the PAP; the application is not aware that they are there.

Opening Brace Execution

The opening brace results in the compiler emitting an *xbegin* instruction. Just after the *xbegin* instruction, a call to Violet's *start_tx* is inserted by the PAP. *start_tx* allocates a Violet transaction descriptor that is used to track the change-set of the transaction. A pointer to the transaction descriptor is placed on stack.

While every assignment between the braces is included in the transaction, not all are relevant to an application's persistent state. For instance, assignments to the stack are irrelevant. Only assignments to memory locations in the mapped file need to be replicated, and these are easily detected as they fall in the address range of the mapped region. The identification of the relevant change-set is done dynamically at run-time with instrumentation inserted at compile-time by the PAP. The PAP inserts an invocation to the *add_write_set* function before the assignments.

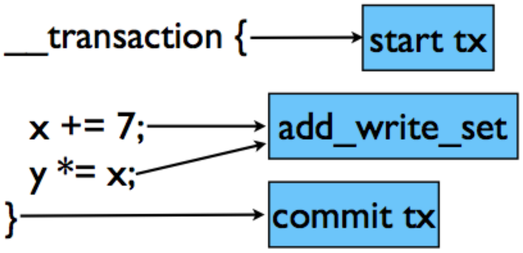


Figure 2: *A Violet transaction. The blue boxes are instrumentation added by Violet.*

Every memory assignment in the transaction block is passed through to *add_write_set* along with the pointer to the transaction descriptor. *add_write_set* identifies relevant addresses with a range check, and then records them in the transaction descriptor. Assignments in the x86 instruction set are of the form *movX*, where *X* is the type/size of the assignment. The type of the *mov* instruction is used by the PAP to determine the size of the assignment. Standard libc functions, such as memmove and strcpy, are not currently supported. Such functions are typically written in hand-tuned assembly and need to be made transactional. Moreover, the maximum size of a TSX change-set is finite and varies with many factors. Bulk data movement is probably best handled with different methods. If demand dictated then Violet versions of the functions could be included in Violet's run-time library.

Closing Brace Execution

A closing brace connotes a transaction commit and the compiler emits a *xend* instruction. The PAP will insert a call to Violet's *commit_tx* **outside** of the transaction block. A restriction of Intel RTM is that system calls (among other things) automatically abort transactions. *commit_tx* makes network system calls so it has to be called outside of the transaction block. This is a window of failure. The window of failure is on the order of a network round-trip-time. The RTM commits the transaction before it is replicated and made durable. Further, a successful transaction is also visible to other threads before it is made durable. Clearly, Violet is currently not suitable for applications that require ACID. We plan to address this in future work.

There are many ways to address the window of failure, such as a introducing some form of pre-logging, but we wanted to avoid imposing onerous burdens on the programmer or violating the C++ standard. To make the system as natural as possible to program, it was decided to identify the change-set automatically at run-time; the window of failure is the trade-off. Violet's window of failure is much smaller than the typical 30-second period between cleaning dirty pages in a buffer cache.

Cleanup of an aborted transaction is trivial. As no memory modifications made inside the transaction block are visible on abort, all of Violet's transaction metadata is cleaned up as a side effect of the failed transaction. While this artifact made implementation easier, it also made debugging difficult.

Replication

Following a successful RTM transaction, Violet replicates the results to a remote storage server. *commit_tx* is responsible for performing replication. The mutations to the file are recorded at the physical level, that is, the result of a transaction is recorded as the set of changed memory locations and their new contents.

An update to memory is encapsulated as a patch. A patch is a tuple consisting of a memory address, length and a string of bytes. Every memory location modified in a transaction is encapsulated in a patch. There is no type information.

The file is sharded over the replication group. The object of a replication group is to put more disks at the disposal of an application to increase disk bandwidth. Multiple disks do not increase reliability with multiple copies; they are there to increase disk bandwidth.

While there is a window of failure, Violet does guarantee that the remote copy of the file is always consistent. Consistency is enforced by the replication group. Memory updates are propagated to the replication group with a 2-phase commit protocol along with a monotonically increasing per transaction serial number. The serial numbers totally order the transactions. Their use is explained in §4.

The replication facility offers two modes of operation. Replication can either optimize network throughput or minimize the window of failure. If the application wishes to minimize the window of failure, the committing thread will wait for the result of the 2-phase commit before proceeding. It will also learn of any errors synchronously.

Far better use of the network is made if the committing thread simply queues its updates for transmission and carries on. The updates are transmitted when sufficient data have accumulated to fill an Ethernet frame. On a busy multi-core system, where transactions take on the order of 100 nanoseconds to complete, the wait time is usually sub-μs. We demonstrate in the results section that the window of failure is virtually the same in both modes of operation when commodity Ethernet is used.

4. Capacity Tier

In this section we describe the storage nodes (Sponges) used to persist the updates to the file on the host. We present a data structure, the Fibonacci Array, that is used to represent the file on disk. Finally, we show how the Sponges create distributed snapshots.

The capacity tier of Violet is comprised of a set of cooperating machines running a software agent called Sponge. It is a user level process. A Sponge is assumed to have at least one locally attached disk and some NVM (non-volatile memory) at its disposal. The Sponge is responsible for providing the capacity tier functions of disaster recovery and data management.

When a Sponge starts it determines what resources are available to it, such as the number of disks, and then it registers with the master server. The master server is discovered with mDNS. The master incorporates the Sponge into the cluster. Once a Sponge is registered with the master it is eligible for assignment to a file's replication group. Sponges can be members of any number of replication groups.

Mating the Host Update Behavior and Disks

A significant challenge for a disk based capacity tier is to mate the expected update behavior of the application on the host with the mechanical block-based world of the disk. Most applications today are conscious that they are backed by mechanical media and take great pains to interact well with them. DRAM data structures make no such concessions and focus on an asymptotic goal; DRAM data structures did not have disks in mind when they were developed. The updates to DRAM data structures can be `tiny' and there can be little locality. Furthermore, as applications are running at memory-bus speed, the rate of updates can be much higher.

In-memory applications only read the data on disk in failure scenarios, and then they stream the entire file. In the steady-state, the workload is write-only. Therefore, an index into the data is not required. An in-memory backing store just needs to build an image of the file from the incoming memory updates while keeping the cost of getting a memory update into the correct position in the file low.

Partial updates and poor locality are not new problems for storage systems. One technique of reducing the cost of an update is to amortize the cost of the I/O over multiple updates. An NVM staging area can be used to absorb updates and order them to detect opportunities for amortizing writes. Unfortunately, in the presence of very poor locality this strategy is not feasible. However, if the staging area is written out in its entirety and contiguously, then excellent amortization is realized; we call this staging with serial writing. In this way the random input stream is inflected into an ordered stream that leverages the streaming bandwidth of disks. Indeed, one can just keep doing this ad infinitum, but at some point the many disk resident logs need to be coalesced and reconciled rationally. A hierarchy must be superimposed on the disk resident pieces to regulate log resolution.

Two examples of disk log hierarchies are the LSM-Tree [12] and the COLA [2]. Both are excellent at dealing with random updates as they employ staging with serial writing. They also continually merge the disk resident pieces to incorporate updates yielding larger pieces. Both are optimized to keep an index into the data up to date. Because LSM-Trees and COLAs need to support lookup operations, the ratio of sizes between successive levels in their hierarchies is fixed to preserve properties required for search efficiency described below. In the absence of this requirement the ratio between levels can be varied permitting the merging of data through the hierarchy to its final resting place at lower cost.

Fibonacci Arrays

The Fibonacci Array [FA] is a data structure used to represent an in-memory file on disk. Each in-memory file has its own FA. The FA employs staging with serial write. The structure of an FA is depicted in Figure 3. It consists of an array of pointers to disk resident logs and a buffer, of size B, in NVM. The length of the levels in the array grows as a Fibonacci sequence; this is not by design but a natural artifact of the merging rule that is described below. The entries in an FA are patches containing memory updates.

The Fibonacci Array is derived from the COLA owing to its low amortized write cost. The array of levels in a COLA is managed like a binary number. The state of the array is a binary string where 1 indicates that a level is occupied and 0 denotes empty. For example, 1101 indicates that the first, second and fourth levels are occupied. A level, k, is either empty or contains 2^k entries. Flushing the NVM buffer to disk binary-adds 1 to the string. Merges occur when a binary carry is needed, e.g., flushing the NVM buffer to 1110 will result in a 4-way merge between levels 1-3 and NVM into the fourth level. The resulting string is 0001; so the 4th level is occupied and all beneath it are now empty. The following NVM flush will produce 1001, but no merge. In general, a merge, where k is the highest bit in the carry, requires $k2^k$ comparisons and a $(k + 1)$-way merge. This can lead to an overwhelming spike of I/O and CPU consumption as the height of the COLA grows. The COLA requires its merge rules to maintain invariants required to support efficient $\log_2$ search. The FA is free to use a different merge rule.

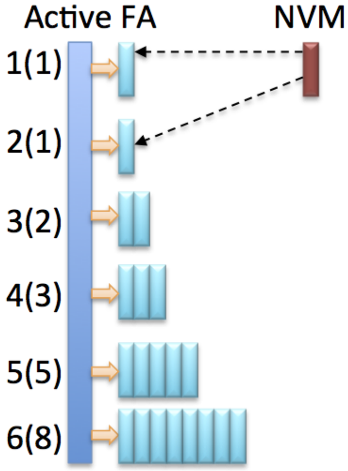


Figure 3: *Structure of a Fibonacci Array. Data start in NVM and then percolate down the array.*

The following rule is used to push data through a FA: when two neighboring levels in the array are occupied then they are merged into the next highest level. The source levels of the merge are left unoccupied. As the size of the merged level is the sum of the two source levels the array is naturally Fibonacci. Also, observe that the lowest source level skipped a level in the array and jumped two levels. Merging is done in the background. As only two levels are merged at any given moment the resource consumption of an FA is much smoother than a COLA. The 2-way merge permits the disk to seek less frequently and makes better use of the CPU.

Data propagate through a FA as follows. Updates arrive from the host in the form of patches that contain memory updates. As patches arrive they are placed in NVM in order of the memory location that they represent. Neighboring and overlapping patches are coalesced into a single patch where possible. As depicted in the diagram, once the NVM buffer is full it is written to disk. The system alternates between writing the NVM buffer to levels 1 and 2. When both are occupied they trigger the merging condition. As the logs are ordered, merging is trivially effected by streaming both source levels into the target level while maintaining the ordering. Neighboring patches are coalesced into a single entry as they are encountered. Patches can overlap producing a conflict. In this case the lower level's values are used as they are younger. Conflicting patches are resolved into a single patch. Applying the merge rule successively to all the levels of the array results in the data percolating down the array (driven by data flushing from NVM). At some point the data will arrive at the terminal level.

The terminal level in the array is the size of the file; a log cannot be larger than the file it represents. The terminal level will converge to a single patch (the length of the file). The height of the array, h, is approximately $\log_{1.61803}$(size of file), where 1.61803, the Golden Ratio, is the ratio between successive Fibonacci numbers. Thus, the per byte amortized cost of getting a byte to the terminal level is the number of times it was written over the size of the write: $0.75h/B$. The factor of 0.75 accounts for the fact that a datum skips a level on every second merge.

Fibonacci Array Snapshots

The FA includes a snapshot mechanism to support replication group snapshots. A FA snapshot is not exposed or created directly. It is the mechanism upon which consistent replication group snapshots are implemented.

A snapshot is created by copying the array of pointers into a new array that will become the active FA. Arrays and logs include a version number. When the new active FA is created its version number is incremented. As data percolate through the system and merging takes place, the mismatch between log version numbers and array version numbers is detected. A new log for the level is created with the active version number, and the pointer in the active array is updated.

FA snapshots are efficient because the active FA diverges from a snapshot at the granularity of the byte - exactly how the file actually is diverging. Explicit COW at the granularity of a page is supplanted by the temporal relationship between levels in the array. If a byte is updated and inserted into the active FA, it will enter the FA at the lowest level. Thus, the fact that it is the valid value in the active FA is implicit, and we do not need to know anything about what it over-wrote or where it is; nothing special has to be done as it is an intrinsic property of the data structure. This is efficient in both space and I/O.

Replication Group Snapshots

Snapshots can be taken of in-memory files stored in a Violet replication group. They offer a consistent view of memory as it appeared to the replication group when the snapshot was taken. An in-memory file stored in a replication group is distributed over more than one machine. A consistent snapshot therefore requires coordinating all of the Sponges in a replication group.

To create a snapshot of an in-memory file, the snapshot requestor queries the cluster master to discover the file's replication group. The requestor will then contact the Sponge with the highest IP address – this will be the leader for the snapshot. The leader contacts all of its peers and informs them of the snapshot request.

The replication group must come to a consensus on when the snapshot took place to ensure that the snapshot contains a consistent view of memory. Consistency is effected with the serial numbers of transactions. A serial number is agreed to such that every transaction that took place prior to it is in the snapshot and all those following are not. This serial number is the *snap-point*.

The host-side library aids in the determination of the snap-point. Whenever it sends a message to a Sponge it includes the latest *consistency-point*. The consistency-point is the highest serial number of a transaction such that there is an unbroken sequence of committed serial numbers back to zero. The leader solicits the highest consistency-point that has been observed by its peers. The highest consistency-point observed by any Sponge in the replication group is chosen as the snap-point. Note that during the course of this algorithm higher consistency-points may arrive on a Sponge, but they are ignored for the purposes of the current attempt of creating a snapshot. This ensures progress.

Once the snap-point has been published, Sponges proceed to create the FA snapshot. All patches in NVM that are in the snapshot are sent to disk. The FA snapshot is then created and the leader is informed. Once the leader has been informed of completion by all of its peers it informs the requestor.

5. Master Server

The master server is the glue that binds all of the pieces together. It is responsible for provisioning a file's replication group and file directory services. There is only one instance of a master in the system. While the master is a single point of failure there are many well known techniques to address high availability, e.g. a committee of machines running Paxos. As Violet is a research system the simplest implementation was adopted. Moreover, the master is not in the data-path and received little attention in this paper.

6. Results

We have implemented and evaluated Violet. The host-side library and tool chain were implemented on MacOS and Linux x86. The Sponge was implemented on

Linux x86. We used our prototype to evaluate the viability and effectiveness of Violet with commodity networks and disks.

The Violet data-path consists of two pieces: the host-side library and the Sponge. The master is not in the data-path and purely ancillary so it is not included.

The evaluation of Violet is decomposed into four parts. First, we measure the overhead of Violet instrumentation in application code to show that it is not onerous. Second, we explore how Violet interacts with a commodity Ethernet network when replicating. Third, we show that the Sponges efficiently create snapshots and restore data in the event of disaster recovery. Finally, we show that Violet can be used to implement DRAM data structures that are superior to standard storage data structures.

Instrumentation Overhead

We evaluated the overhead of Violet instrumentation on an Apple iMac configured with 16GB of DRAM. The processor is a 4-core Haswell i7-4771 with a clock speed of 3.5GHz. The i7-4771 supports TSX and hyper-threading.

Violet anticipates a new generation of in-memory DBs. As such there are no extant benchmarks or applications that are appropriate for the evaluation of Violet (nobody persists a C++ std::map today). As Violet was conceived for DRAM data structures we used two of the most common such data structures for Violet's evaluation: a self-balancing binary tree (a probabilistically balanced tree called a Treap) and a linked list.

To measure the cost of Violet's instrumentation a Treap was implemented with the proposed C++ TM standard. We measured its performance and compared it to the same code after the PAP was used to instrument it. Figure 4 presents the results.

The curve labeled Violet is a fully instrumented Treap. The RTM curve is the same code, but without Violet instrumentation. The times reported are the times taken to insert a single element in the Treap. The difference between the curves on the Δt axis is the temporal overhead introduced by Violet.

The initial knee in the curve results from going from a single thread to multiple threads. The time to insert an element remains roughly constant between 2 and 8 threads, as does the difference between the curves. In this range there is little pre-emption and CPU affinity is high. As the number of threads grows beyond 8 they start to compete for CPU time and pre-emption becomes a factor. Pre-emption causes RTM transactions to abort and the number of transaction retries begins to climb; this behavior is very different from locking where locks are held across pre-emption.

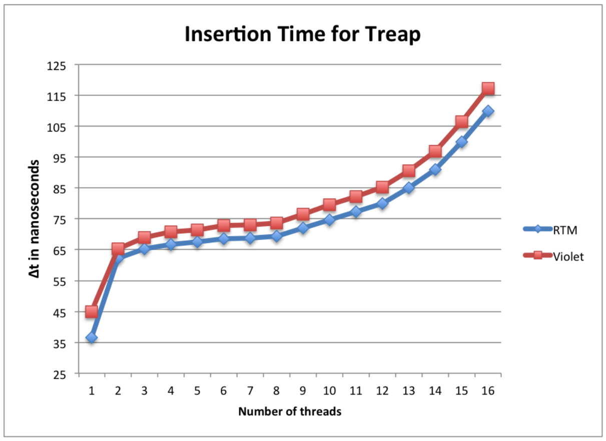


Figure 4: *Elapsed time for Treap insertions. Note that system call over-head is 200ns so far too slow for transactions on this time scale.*

The overhead of Violet's instrumentation appears to be constant, and quite low. Violet's instrumentation is very simple: two integer comparisons that comprise the address range check. The two integers that comprise the range are in the same cache-line and so popular that they are usually in the cache. Most of the expense of a transaction is incurred by loading the transaction's read-set from memory. All memory writes in a transaction are confined to the L1 cache until the transaction is committed.

Replication Performance

In this section we examine Violet's replication performance over commodity Ethernet. Applications that require replication but wish to run at CPU speeds will be highly sensitive to network performance.

A challenge when evaluating a system with a hardware dependency is gaining access to the hardware. The iMac is our sole TSX platform, but its only Ethernet option is 1 Gb. We felt that 1 Gb is too unrealistic for a modern server; a single Violet thread can saturate the iMac's NIC.

To perform experiments with 10 Gb Ethernet we used Amazon EC2. The EC2 `cluster instance' provides 10 Gb Ethernet, 2 Xeon processors (for a total of 8 cores), 1 disk and 60 GB of RAM. The `cluster instance' does not share processors; the processors are dedicated, but they do not support TSX.

The following experiments did not use TSX. Solely for this experiment the PAP removed the RTM instructions, but it produced fully instrumented code (that was not thread-safe). Threads were given exclusive access to private Treaps for correctness. The experimental setup consisted of a single host running the Violet Treap application replicating to a replication group that was configured with eight Sponges.

Figure 5 presents the aggregate throughput to the replication group as a function of the number of threads. We show two curves. The EC2 curve is the Treap application without RTM. This was run in EC2 and actually performed replication. The TSX curve is data produced by the iMac running fully instrumented RTM code, but not replicating (so just going as fast as it could). We include it to show that when Violet uses RTM it is still capable of driving an 8-node replication group.

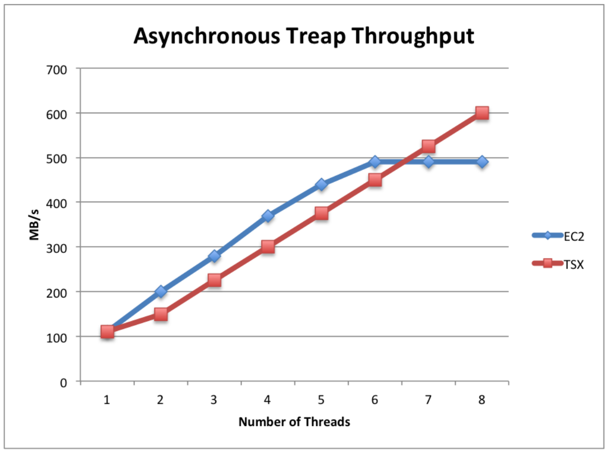


Figure 5: *Application insertion throughput*.

EC2 throughput increases close to linearly until 490 MB/s is reached. This occurs when 6 or more threads are used. Benchmarking revealed that the EC2 10 Gb network is only capable of 490 MB/s so this is the saturation point of the network. If a superior network was available we believe that the throughput would have continued to grow because the Sponges had not yet reached their limit. The TSX curve intersects the EC2 curve where the latter hits the network's saturation point. This suggests that if the EC2 machines had been using RTM then the point of network saturation would have been postponed, as it would have been slightly slower.

The next experiment measures the performance of replicating synchronously. To demonstrate the sensitivity of replication to the choice of data structure an ordered doubly linked list was also used. Figure 6 presents the throughput for list updates being replicated both synchronously and asynchronously. The difference between the two is a factor of 2. Figure 6 also includes Treaps replicating synchronously. Asynchronously a single Treap thread produces ≈100 MB/s (Figure 5), but synchronously it slows down to 0.3 MB/s. This is a manifestation of the mismatch between the network's round-trip-time [RTT] and the time taken to commit a RTM transaction. The RTT in the EC2 network is 220 µs, and the time to execute a transaction is 45ns-120ns. A single transaction consists of only 40-80 bytes so the high latency is exacerbated by sending Ethernet frames that are practically empty. Lists do not produce as many updates as Treaps, threads spend more time looking for the insertion point in a list, so lists are not as sensitive as Treaps to synchronous replication.

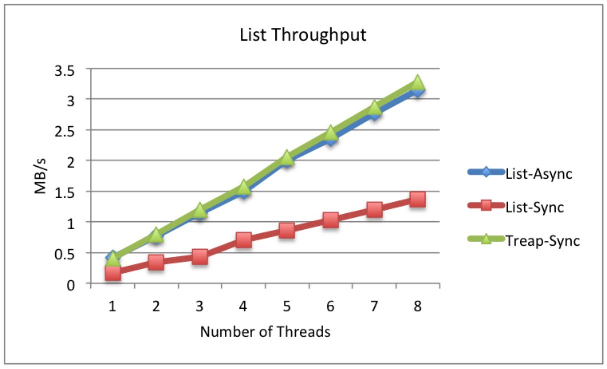


Figure 6: *List and Treap replication throughput*.

In the time it takes a Treap thread to synchronously replicate one transaction it could have committed ≈5,500 transactions. Moreover, 5,500 transactions would have filled many Ethernet frames in the same time frame thus triggering their transmission. We found that asynchronously replicated patches never waited longer than 1µs to be sent and on average waited 200ns. Thus asynchronously replicating updates offers roughly the same window of failure while dramatically increasing throughput. The throughput of synchronous replication is not viable over commodity Ethernet. For applications that can tolerate a small window of failure, we believe that Violet running with asynchronous replication is compelling. It is not uncommon for applications to tolerate dirty pages sitting in a kernel's buffer cache for 10's of seconds suggesting such applications exist.

Snapshot Creation Analysis

Snapshot performance as a function of the number of machines is depicted in Figure 7. A 25GB file was populated and then snapped. Taking a snapshot is quick operation requiring less than a second. The knee in the curve going from 1 to 2 machines results from consulting peer machines. A single machine can trivially find the snap point. When more than one machine is involved then the snap point must be found by inter-machine communication. Despite the knee in the curve, a replication group can create a snapshot in less than a second.

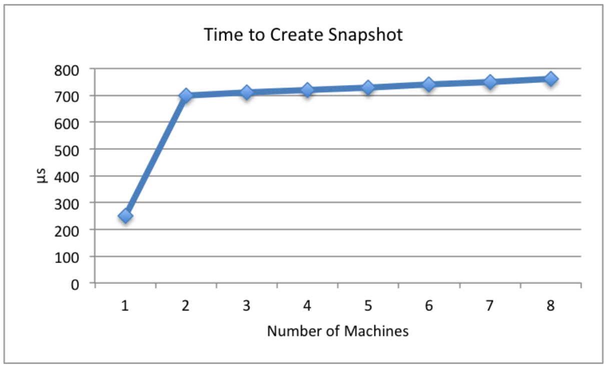


Figure 7: *Time required to take snapshot as observed by requestor***.**

Restore Analysis

To be useful for disaster recovery the system has to be able to restore a file in a sufficiently `short' period of time. In an environment such as EC2 the fastest one can restore a file is constrained by the network connection between the machines. The top throughput of EC2's network is 490 MB/s. The question then becomes how many Sponges does it take to saturate the network (this is the time to restore).

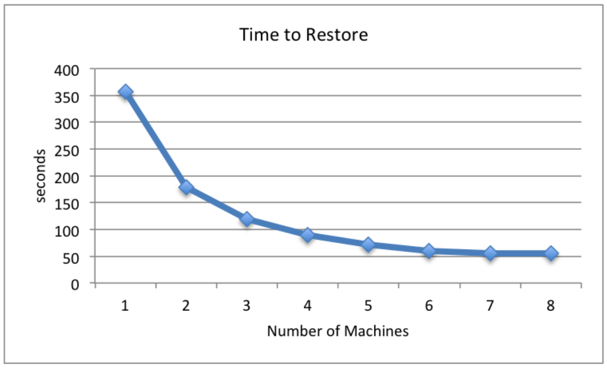


Figure 8: *Time to restore as a function of Sponges/disks***.**

Restore performance is presented as a function of the number of machines in Figure 8. The size of the file is 25 GB. As can be seen from the graph, the Sponges saturate the network at 6 machines. When the network is saturated it takes 53 seconds to restore a 25GB file. The individual Sponges are always disk bound and manage ~80 MB/s each until the network is saturated at 6 machines. This is close to the peak read rate observed in the cluster instance (90 MB/s). The contiguous disk layout of the FA saves the disk from seeking frequently.

Asymptotic Driven Data Structures

In-memory computing has vastly increased the speed of analytics. However, much of the advancement has come from the improvement of the inherent characteristics of the media, by using DRAM instead of disks. Merely swapping the media type only partially realizes the potential speed-up if the same data structures continue to be used. For example, SAP HANA is an in-memory DB, but for the most part it is columnar. This is not very different from deploying a columnar DB on a RAM disk. Merely exchanging media ignores many of the other advantages of DRAM; e.g., it offers random byte-grained access. The adoption of DRAM suggests that different data structures could be used thus also realizing an *asymptotic* speed improvement.

The greatest potential for performance improvement of in-memory DBs lies in adopting data structures that are usually overlooked because they are difficult to persist: DRAM data structures. DRAM is volatile so in-memory databases must still persist their state to block devices. An obstacle to the adoption of more complicated data structures is the difficulty in persisting them correctly and efficiently. Violet was developed to bridge this gap.

Employing domain specific databases has been suggested in the past [8, 9] and shown to be superior. To motivate this argument we present two queries that are important to our customers that they typically run on columnar DBs. For our experiment both queries were run on MonetDB [3] provisioned with a RAM disk. The queries were also run on a domain specific data structures created with Violet. The point is not to demonstrate that the Violet system is faster per se (it is not a fair comparison), but to demonstrate how poor linear scans are for many problems important to our customers. If databases are produced that can cut hours off of computation they will be adopted [8, 9].

The first query that we present is the identification of clusters of points in a data set. Cluster identification is an important knowledge discovery technique, e.g. market researchers interpret the clusters as market segments. An important algorithm for this application is DBScan [4]. DBScan runs in $O(N^2)$ time with a columnar DB, but it runs in $O(N)$ time when the DB supports a nearest neighbor query. The Voronoi diagram supports the nearest neighbor query so it was implemented with Violet. The results are shown in Figure 9; the predicted difference in growth of run-times is observed (note the log scale).

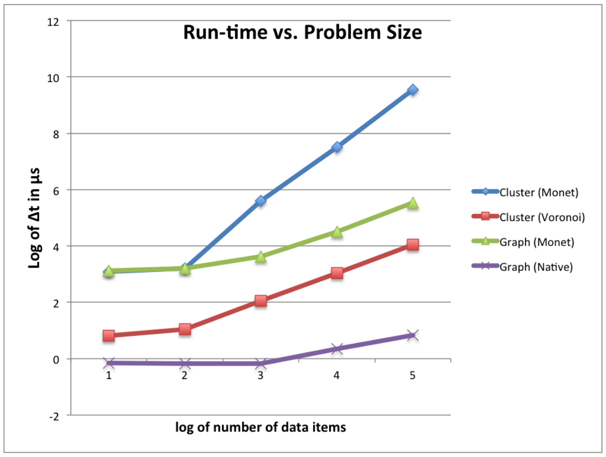


Figure 9: *log/log scale asymptotic behaviour of clustering and path queries***.**

The second query that we examined is a path query in a graph. Graphs arise naturally in a lot of applications and they have been attracting a lot of attention in analytics; e.g., SAP is embracing graph computing by grafting a graph query engine on to HANA [17]. A columnar DB, however, is not the most natural way to represent a graph in memory. We implemented a graph DB that represents the graph as memory nodes and pointers. Queries on a graph take *O(N)* time in a columnar DB, but only grow by the diameter of the graph when represented naturally. Figure 9 presents the observed empirical asymptotic superiority of the natural representation.

In both implementations of the domain optimal data structures there is a gross asymptotic advantage hence a strong motivation to adopt them. Violet made it easy to persist the data structures as no thought had to be given to the persistence layer; a non-trivial problem in the case of a Voronoi diagram. Violet just took care of it. Implementation effort was directed at a correct implementation of the data structure.

7 Related Work

SNIA non-volatile memory (NVM) working group has proposed a standard for accessing new types of non-volatile memories [16]. Currently, they have proposed both a kernel level volume, and a kernel level file interface for accessing NVM at a fine-grained level. They are also interested in proposing a user space level interface to allow applications to access NVM at a fine-grained level using the load and store data access model. They also want to introduce the notion of transactions for this user level API. The work that has been proposed in this paper can be leveraged by this SNIA working group.

Different data management middleware offerings like Redis [15], SAP HANA [11], Microsoft_Grace [19], Facebook_TAO [20] are proposing strategies for data management at the data structure level (e.g. graphs, KV stores). Redis allows for the persistent management of key-value storage data structures. Redis provides off-node data protection by copying the data at a file level. SAP Hana allows for in-memory manipulation of graph data structures and it maps these data structures on to a columnar database that, in turn, moves data off-node at page level granularity [17]. Microsoft Grace system stores the graph data structures such as vertices and edges in respective files on disk, and subsequently it reads these structures in parallel when loading in the graph. Additionally, Grace also maintains a log of committed updates on disk. Facebook TAO is a geographically distributed eventually consistent graph store. TAO shards the dataset into shards and stores these shards across multiple database servers. TAO also maintains an elaborate leader-follower caching infrastructure in front of its persistence layer, and it uses an eventual consistency model to keep the data consistent amongst the caches.

Recoverable Virtual Memory [22] addresses a similar space as Violet, but takes a different approach. RVM employs explicit logging and requires the programmer to identify and backup copies of data. Concurrency is consciously left to the programmer to address separately. The file is updated with a naïve staging-and-write strategy. Consistency in the file is maintained with the log and write-ordering the file's dirty pages.

The work being proposed in this paper is independent of the type of data structure being supported in memory. That is, we detect updates to any type of data structure at a fine granularity and then subsequently ship these fine-grained updates off node and stream them on to a block based back-end storage device. Thus, our work can be leveraged by the above mentioned middleware systems.

8 Conclusions

In this paper we describe a storage system that spans across the host and a disk-based backend storage system. This architecture helps to map fine-grained host side data management with a block level data management system at the backend. The techniques presented in this paper become important as real-time analytics applications begin to employ data structures that have been designed with main-memory computing model in mind and that want to backup these data structures in an efficient manner on to a cheaper off-box disk based storage system. We propose a host side user space client library that leverages the CPU's transactional memory instructions to efficiently detect fine-grained updates. We also present a new data structure called a Fibonacci Array at the backend disk subsystem that helps to stream update operations on to a disk in a more efficient manner by optimizing the data structure for primarily write operations. The Violet system divorces the implementation of a data structure from its persistence. Finally, we implemented our ideas in a prototype and demonstrated the benefits of managing in-memory data structures natively, rather than mapping them to intermediate data stores that are not designed to deal with in-memory data structures natively.

Bibliography

[1] F. Aurenhammer. "Voronoi diagrams: A survey of a fundamental geometric data structure." In ACM Computing Surveys, 23(3):345–405, 1991

[2] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. "Cache-oblivious streaming B-trees." In SPAA, 2007

[3] Peter Boncz, Stefan Manegold and Martin Kersten, "Database Architecture Optimized for the New Bottleneck: Memory Access", In VLDB, 1999

[4] M. Ester, Hans-Peter et al (1996). "A density-based algorithm for discovering clusters in large spatial databases with noise", Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (1996).

[5] Rich Freitas and Larry Chiu, "Storage Class Memory: Technologies, Systems, and Applications." In USENIX FAST 2012 Tutorials

[6] Zhaoguo Wang, Hao Qian et al, "Using Restricted Transactional Memory to Build a Scalable In-Memory Database," Eurosys 2014

J.Guerra, L.Marmol et al, "Software Persistent Memory," In USENIX ATC 2012

[7] M. Herlihy and J. E. B. Moss. "Transactional memory: Architectural support for lock-free data structures." In Annual International Symposium on Computer Architecture, 1993.

[8] M. Stonebraker and U. Cetintemel. "One Size Fits All: An Idea whose Time has Come and Gone." In ICDE 2005.

[9] M. Stonebraker, Nabil Hachem et al., "The End of an Architectural Era (It's Time for a Complete Rewrite)." In VLDB 2007

[10] M. Stonebraker, "New SQL: An Alternative to NoSQL and Old SQL for New OLTP Apps", in Communications of the ACM, June, 2011.

[11] F. Farber, S. Cha, J. Primsch, C. Bornhovd, S. Sigg W. Lehner. "SAP HANA Database: Data Management for Modern Business Applications", in SIGMOD Record, January, 2012.

[12] O'Neil, P. E., Cheng et al. "The log-structured merge-tree (LSM-Tree)." in Acta Informatica 33, 1996

[13] "Oracle TimesTen In-Memory Database on Oracle Exalogic Elastic Could", Oracle White Paper, July, 2011.

[14] "VoltDB: High Performance, Scalable RDBMS for Big Data and Real-Time Analytics", White Paper, 2012.

[15] Jeremy Zawodny, "Redis: Lightweight key/value Store That Goes the Extra Mile", Linux Magazine, August 31, 2009

[16] SNIA NVM Programming Model, version 1.0.0 Revision 5, Working Draft, June 12, 2013

[17] M. Rudolf, M. Paradies, C. Bornhovd, and W. Lehner. "The Graph Story of the SAP HANA Database", in BTW, 2013.

[18] Intel® 64 and IA-32 Architectures Optimization Reference Manual, Document Number: 248966-028 July 2013

[19] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou and M. Haridasan, "Managing Large Graphs on Multi-Cores with Graph Awareness", USENIX ATC, 2012.

[20] N. Bronson et al, "TAO: Facebook's Distributed Data Store for the Social Graph", USENIX ATC, 2013.

[21] Transactional Memory Support for C++, www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3859.pdf

[22] M. Satyanarayanan et al, "Lightweight Recoverable Virtual Memory," SOSP 1993

[23] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In Proc. ICDE, 2014.

ELF: Efficient Lightweight Fast Stream Processing at Scale

Liting Hu, Karsten Schwan, Hrishikesh Amur, Xin Chen
Georgia Institute of Technology
{foxting,amur,xchen384}@gatech.edu, schwan@cc.gatech.edu

Abstract

Stream processing has become a key means for gaining rapid insights from webserver-captured data. Challenges include how to scale to numerous, concurrently running streaming jobs, to coordinate across those jobs to share insights, to make online changes to job functions to adapt to new requirements or data characteristics, and for each job, to efficiently operate over different time windows.

The ELF stream processing system addresses these new challenges. Implemented over a set of agents enriching the web tier of datacenter systems, ELF obtains scalability by using a decentralized "*many masters*" architecture where for each job, live data is extracted directly from webservers, and placed into memory-efficient *compressed buffer trees* (CBTs) for local parsing and temporary storage, followed by subsequent aggregation using *shared reducer trees* (SRTs) mapped to sets of worker processes. Job masters at the roots of SRTs can dynamically customize worker actions, obtain aggregated results for end user delivery and/or coordinate with other jobs.

An ELF prototype implemented and evaluated for a larger scale configuration demonstrates scalability, high per-node throughput, sub-second job latency, and sub-second ability to adjust the actions of jobs being run.

1 Introduction

Stream processing of live data is widely used for applications that include generating business-critical decisions from marketing streams, identifying spam campaigns for social networks, performing datacenter intrusion detection, etc. Such diversity engenders differences in how streaming jobs must be run, requiring synchronous batch processing, asynchronous stream processing, or combining both. Further, jobs may need to dynamically adjust their behavior to new data content and/or new user needs, and coordinate with other concurrently running jobs to share insights. Figure 1 exemplifies these requirements, where job inputs are user activity logs, e.g., *clicks*, *likes*, and *buys*, continuously generated from say, the *Video Games* directory in an e-commerce company.

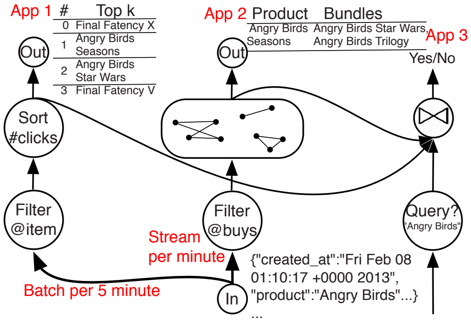


Figure 1: Examples of diverse concurrent applications.

In this figure, the `micro-promotion` application extracts user *clicks* per product for the past 300 *s*, and lists the top-*k* products that have the most clicks. It can then dispatch coupons to those "popular" products so as to increase sales. A suitable model for this job is synchronous batch processing, as all log data for the past 300 *s* has to be on hand before grouping clicks per product and calculating the top-*k* products being viewed.

For the same set of inputs, a concurrent job performs `product-bundling`, by extracting user *likes* and *buys* from logs, and then creating 'edges' and 'vertices' linking those video games that are typically bought together. One purpose is to provide online recommendations for other users. For this job, since user activity logs are generated in realtime, we will want to update these connected components whenever possible, by fitting them into a graph and iteratively updating the graph over some sliding time window. For this usage, an asynchronous stream processing model is preferred to provide low latency updates.

The third `sale-prediction` job states a product name, e.g., *Angry Birds*, which is joined with the product-bundling application to find out what products are similar to *Angry Birds* (indicated by the 'typically bought together' set). The result is then joined with the micro-promotion application to determine whether *Angry Birds* and its peers are currently "popular". This final result can be used to predict the likely market success of launching a new product like *Angry Birds*, and obtaining it requires interacting with the first and second application.

Finally, all of these applications will run for some considerable amount of time, possibly for days. This makes it natural for the application creator to wish to update job functions or parameters during ongoing runs, e.g., to change the batching intervals to adjust to high vs. low traffic periods, to flush sliding windows to 'reset' job results after some abnormal period (e.g., a flash mob), etc.

Distributed streaming systems are challenged by the requirements articulated above. First, concerning *flexibility*, existing systems typically employ some fixed execution model, e.g., Spark Streaming [28] and others [11, 12, 17, 22] treat streaming computations as a series of batch computations, whereas Storm [4] and others [7, 21, 24] structure streaming computations as a dataflow graph where vertices asynchronously process incoming records. Further, these systems are not designed to be naturally composable, so as to simultaneously provide both of their execution models, and they do not offer functionality to coordinate their execution. As a result, applications desiring the combined use of their execution models must use multiple platforms governed via external controls, at the expense of simplicity.

A second challenge is *scaling with job diversity*. Many existing systems inherit MapReduce's "*single master/-many workers*" infrastructure, where the centralized master is responsible for all scheduling activities. How to scale to hundreds of parallel jobs, particularly for jobs with diverse execution logics (e.g., pipeline or cyclic dataflow), different batch sizes, differently sized sliding windows, etc? A single master governing different per-job scheduling methods and carrying out cross-job coordination will be complex, creating a potential bottleneck.

A third challenge is *obtaining high performance for incremental updates*. This is difficult for most current streaming systems, as they use an in-memory *hashtable*-like data structure to store and aggressively integrate past states with new state, incurring substantial memory consumption and limited throughput when operating across large-sized history windows.

The ELF (**E**fficient, **L**ightweight, **F**lexible) stream processing system presented in this paper implements novel functionality to meet the challenges listed above within a single framework: to efficiently run hundreds of concurrent and potentially interacting applications, with diverse per-application execution models, at levels of performance equal to those of less flexible systems.

As shown in Figure 2, each ELF node resides in each webserver. Logically, they are structured as a million-node overlay built with the Pastry DHT [25], where each ELF application has its own respective set of master and worker processes mapped to ELF nodes, self-constructed as a *shared reducer tree* (SRT) for data aggregation. The system operates as follows: (1) each line of logs received from webserver is parsed into a key-value pair and continuously inserted into ELF's local in-memory *compressed buffer tree* (CBT [9]) for pre-reducing; (2) the distributed key-value pairs from CBTs "roll up" along the SRT, which progressively reduces them until they reach the root to output the final result. ELF's operation, therefore, entirely bypasses the storage-centric data path, to rapidly process live data. Intuitively, with a DHT, the masters of different applications will be mapped to different nodes, thus offering scalability by avoiding the potential bottleneck created by many masters running on the same node.

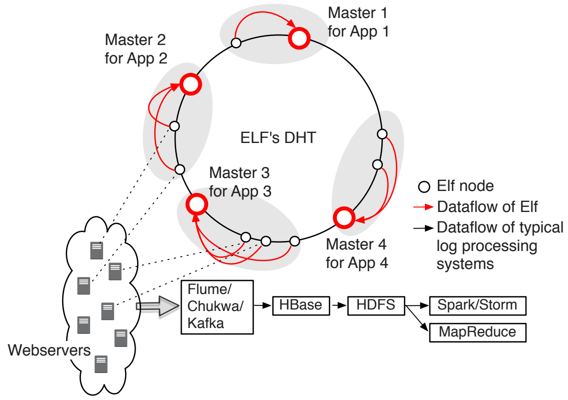


Figure 2: Dataflow of ELF vs. a typical realtime web log analysis system, composed of Flume, HBase, HDFS, Hadoop MapReduce and Spark/Storm.

ELF is evaluated experimentally over 1000 logical webservers running on 50 server-class machines, using both batched and continuously streaming workloads. For batched workload, ELF can process millions of records per second, outperforming general batch processing systems. For a realistic social networking application, ELF can respond to queries with latencies of tens of millisecond, equaling the performance of state-of-the-art, asynchronous streaming systems. New functionality offered by ELF is its ability to dynamically change job functions at sub-second latency, while running hundreds of jobs subject to runtime coordination.

This paper makes the following technical contributions:

1. A decentralized 'many masters' architecture assigning each application its own master capable of individually controlling its workers. To the best of our knowledge, ELF is the first to use a decentralized architecture for scalable stream processing (Sec. 2).
2. A memory-efficient approach for incremental updates, using a *B*-tree-like in-memory data structure, to store and manage large stored states (Sec. 2.2).
3. Abstractions permitting cyclic dataflows via feedback loops, with additional uses of these abstractions including the ability to rapidly and dynamically change job behavior (Sec. 2.3).
4. Support for cross-job coordination, enabling interactive processing that can utilize and combine multiple jobs' intermediate results (Sec. 2.3).
5. An open-source implementation of ELF and a comprehensive evaluation of its performance and functionality on a large cluster using real-world webserver logs (Sec. 3, Sec. 4).

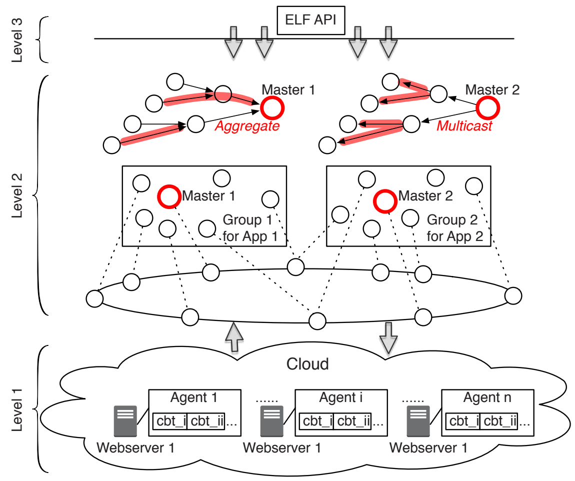


Figure 3: High-level overview of the ELF system.

2 Design

This section describes ELF's basic workflow, introduces ELF's components, shows how applications are implemented with its APIs, and explains the performance, scalability and flexibility benefits of using ELF.

2.1 Overview

As shown in Figure 3, the ELF streaming system runs across a set of agents structured as an overlay built using the Pastry DHT. There are three basic components. First, on each webserver producing data required by the application, there is an agent (see Figure 3 bottom) locally parsing live data logs into application-specific *key-value* pairs. For example, for the micro-promotion application, batches of logs are parsed as a map from product name (key) to the number of clicks (value), and groupby-aggregated like $\langle(a,1),(b,1),(a,1),(b,1),(b,1)\rangle \rightarrow \langle(a,2),(b,3)\rangle$, labelled with an integer-valued timestamp for each batch.

The second component is the middle-level, application-specific group (Figure 3 middle), composed of a master and a set of agents as workers that jointly implement (1) the *data plane*: a scalable aggregation tree that progressively 'rolls up' and reduces those local key-value pairs from distributed agents within the group, e.g., $\langle(a,2),(b,3)..\rangle,\langle(a,5)..\rangle,\langle(b,2),(c,2)..\rangle$ from tree leaves are reduced as $\langle(a,7),(b,5),(c,2)..\rangle$ to the root; (2) the *control plane*: a scalable multicast tree used by the master to control the application's execution, e.g., when necessary, the master can multicast to its workers within the group, to notify them to empty their sliding windows and/or synchronously start a new batch. Further, different applications' masters can exchange queries and results using the DHT's routing substrate, so that given any application's name as a key, queries or results can be efficiently routed to that application's master (within $O(logN)$ hops), without the need for coordination via some global entity. The resulting model supports the concurrent execution of diverse applications and flexible coordination between those applications.

The third component is the high-level ELF programming API (Figure 3 top) exposed to programmers for implementing a variety of diverse, complex jobs, e.g., streaming analysis, batch analysis, and interactive queries. We next describe these components in more detail.

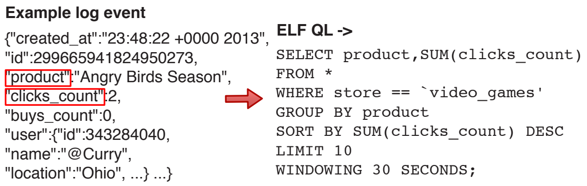
2.2 CBT-based Agent

Existing streaming systems like Spark [28], Storm [4] typically consume data from distributed storage like HDFS or HBase, incurring cross-machine data movement. This means that data might be somewhat stale when it arrives at the streaming system. Further, for most realworld jobs, their '*map*' tasks could be '*pre-reduced*' locally on webservers with the most parallelism, and only the intermediate results need to be transmitted over the network for data shuffling, thus decreasing the process latency and most of unnecessary bandwidth overhead.

ELF adopts an 'in-situ' approach to data access in which incoming data is injected into the streaming system directly from its sources. ELF agents residing in each webserver consume live web logs to produce succinct *key-value* pairs, where a typical log event is a 4-*tuple* of $\langle timestamp, src\_ip, prority, body\rangle$: the *timestamp* is used to divide input event streams into batches of different epochs, and the *body* is the log entry body, formatted as a map from a string attribute name (key) to an arbitrary array of bytes (value).

Each agent exposes a simple HiveQL-like [26] query interface with which an application can define how to filter and parse live web logs. Figure 4 shows how the micro-promotion application uses ELF QL to define the top-*k* function, which calculates the top-10 popular products that have the most clicks at each epoch (30 *s*), in the *Video Game* directory of the e-commerce site.

Each ELF agent is designed to be capable of holding a considerable number of '*past*' key-value pairs, by storing such data in compressed form, using a space-efficient, *in-memory* data structure, termed a *compressed buffer tree* (CBT) [9]. Its *in-memory* design uses an (a,b)-tree with each internal node augmented by a memory buffer. Inserts and deletes are not immediately performed, but buffered in successive levels in the tree allowing better



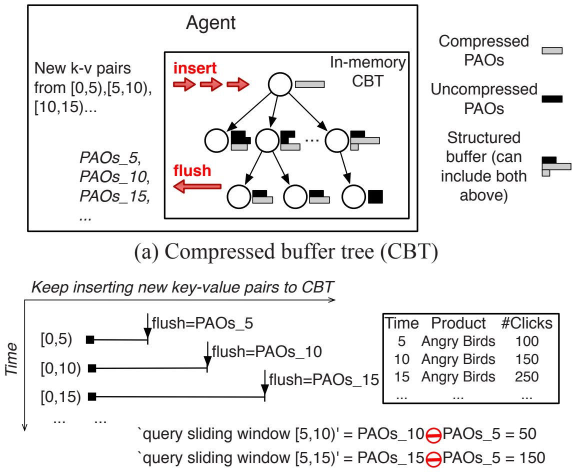
Example log event

```
{"created_at":"23:48:22 +0000 2013",
"id":299665941824950273,
"product":"Angry Birds Season",
"clicks_count":2,
"buys_count":0,
"user":{"id":343284040,
"name":"@Curry",
"location":"Ohio", ...} ...}
```

ELF QL ->

```
SELECT product,SUM(clicks_count)
FROM *
WHERE store == `video_games'
GROUP BY product
SORT BY SUM(clicks_count) DESC
LIMIT 10
WINDOWING 30 SECONDS;
```

Figure 4: Example of ELF QL query.


Agent
New k-v pairs from [0,5),[5,10),[10,15)...
insert
In-memory CBT
flush
PAOs_5, PAOs_10, PAOs_15, ...
...
Compressed PAOs
Uncompressed PAOs
Structured buffer (can include both above)

(a) Compressed buffer tree (CBT)

Keep inserting new key-value pairs to CBT
Time
[0,5)
[0,10)
[0,15)
... ...
flush=PAOs_5
flush=PAOs_10
flush=PAOs_15

Time	Product	#Clicks
5	Angry Birds	100
10	Angry Birds	150
15	Angry Birds	250
...	...	...

`query sliding window [5,10)' = PAOs_10⊖PAOs_5 = 50
`query sliding window [5,15)' = PAOs_15⊖PAOs_5 = 150

(b) Arbitrary sliding windows using CBT

Figure 5: Between intervals, new *k-v* pairs are inserted into the CBT; the root buffer is sorted and aggregated; the buffer is the split into fragments according to hash ranges of children, and each fragment is compressed and copied into the respective children node; at each interval, the CBT is flushed.

I/O performance.

As shown in Figure 5(a), first, each newly parsed key-value pair is represented as a *partial aggregation object* (PAO). Second, the PAO is serialized and the tuple $\langle hash, size, serializedPAO \rangle$ is appended to the root node's buffer, where *hash* is a hash of the key, and *size* is the size of the *serialized PAO*. Unlike a binary search tree in which inserting a value requires traversing the tree and then performing the insert to the right place, the CBT simply defers the insert. Then, when a buffer reaches some threshold (e.g., half the capacity), it is flushed into the buffers of nodes in the next level. To flush the buffer, the system sorts the tuples by hash value, decompresses the buffers of the nodes in the next level, and partitions the tuples into those receiving buffers based on the hash value. Such an insertion behaves like a *B*-tree: a full leaf is split into a new leaf and the new leaf is added to the parent of the leaf. More detail about the CBT and its properties appears in [9].

Key for ELF is that the CBT makes possible the rapid incremental updates over considerable time windows, i.e., extensive sets of historical records. Toward that end, the following APIs are exposed for controlling the CBT: (i) "*insert*" to fill, (ii) "*flush*" to fetch the tree's entire groupby-aggregate results, and (iii) "*empty*" to empty the tree's buffers, which is necessary when the application wants to start a new batch. By using a series of "*insert*", "*flush*", "*empty*" operations, ELF can implement many of standard operations in streaming systems, such as sliding windows, incremental processing, and synchronous batching.

For example, as shown in Figure 3(b), let the interval be 5 *s*, a sale-prediction application tracks the up-to-date *#clicks* for the product *Angry Birds*, by inserting new key-value pairs, and periodically flushing the CBT. The application obtains the local agent's results in intervals [0,5), [0,10), [0,15), etc. as PAO_5, PAO_{10}, PAO_{15}, etc. If the application needs a windowing value in [5,15), rather than repeatedly adding the counts in [5,10) with multiple operations, it can simply perform one single operation $PAO_{15} \ominus PAO_5$, where $\ominus$ is an "invertible reduce". In another example using synchronous batching, an application can start a new batch by erasing past records, e.g., tracking the promotion effect when a new advertisement is launched. In this case, all agents' CBTs coordinate to perform a simultaneous "empty" operation via a multicast protocol from the middle-level's DHT, as described in more detail in Sec.2.3.

Why CBTs? Our concern is performance. Consider using an *in-memory* binary search tree to maintain key-value pairs as the application's states, without buffering and compression. In this case, inserting an element into the tree requires traversing the tree and performing the insert — a read and a write operation per update, leading to poor performance. It is not necessary, however, to aggregate each new element in such an aggressive fashion: *integration can occur lazily*. Consider, for instance, an application that determines the top-10 most popular items, updated every 30 *s*, by monitoring streams of data from some e-commerce site. The incoming rate can be as high as millions per second, but CBTs need only be flushed every 30 *s* to obtain the up-to-date top-10 items. The key to efficiency lies in that "*flush*" is performed in relatively large chunks while amortizing the cost across a series of small "inserting new data" operations: decompression of the buffer is deferred until we have batched enough inserts in the buffer, thus enhancing the throughput.

2.3 DHT-based SRT

ELF's agents are analogous to stateful vertices in dataflow systems, constructed into a directed graph in which data passes along directed edges. Using the terms vertex and agent interchangeably, this subsection describes how we leverage DHTs to construct these dataflow graphs, thus obtaining unique benefits in flexibility and scalability. To restate our goal, we seek a design that meets the following criteria:

1. capability to host hundreds of concurrently running applications' dataflow graphs;
2. with each dataflow graph including minion vertices, as our vertices reside in distributed webservers; and
3. where each dataflow graph can flexibly interact with others for runtime coordination of its execution.

ELF leverages DHTs to create a novel '*many master*' decentralized infrastructure. As shown in Figure 6, all agents are structured into a P2P overlay with DHT-based

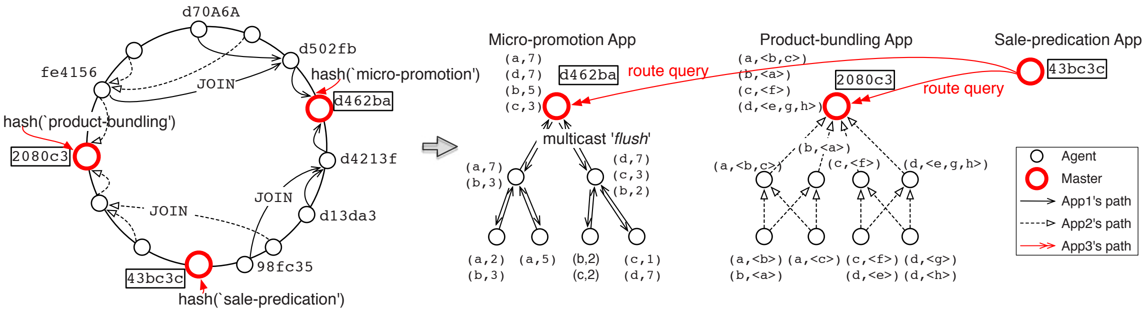


Figure 6: Shared Reducer Tree Construction for many jobs.

routings. Each agent has a unique, 128-*bit* nodeId in a circular nodeId space ranging from 0 to 2^{128}-1. The set of nodeIds is uniformly distributed; this is achieved by basing the nodeId on a secure hash (SHA-1) of the node's IP address. Given a message and a key, it is guaranteed that the message is reliably routed to the node with the nodeId numerically closest to that key, within $\lceil log_{2^b}N \rceil$ hops, where b is a base with typical value 4. SRTs for many applications (jobs) are constructed as follows.

The first step is to construct application-based groups of agents and ensure that these groups are well balanced over the network. For each job's group, this is done as depicted in Figure 6 left: the agent parsing the application's stream will route a JOIN message using appId as the key. The appId is the hash of the application's textual name concatenated with its creator's name. The hash is computed using the same collision resistant SHA-1 hash function, ensuring a uniform distribution of appIds. Since all agents belonging to the same application use the same key, their JOIN message will eventually arrive at a rendezvous node, with nodeId numerically close to appId. The rendezvous node is set as the job's master. The unions of all messages' paths are registered to construct the group, in which the internal node, as the forwarder, maintains a children table for the group containing an entry (IP address and appId) for each child. Note that the uniformly distributed appId ensures the even distribution of groups across all agents.

The second step is to "draw" a directed graph within each group to guide the dataflow computation. Like other streaming systems, an application specifies its dataflow graph as a logical graph of stages linked by connectors. Each connector could simply transfer data to the next stage, e.g., *filter* function, or shuffle the data using a portioning function between stages, e.g., *reduce* function. In this fashion, one can construct the pipeline structures used in most stream processing systems, but by using feedback, we can also create nested cycles in which a new epoch's input is based on the last epoch's feedback result, explained in more detail next.

Pipeline structures. We build aggregation trees using DHTs for pipeline dataflows, in which each level of the tree progressively '*aggregates*' the data until the result arrives at the root. For a non-partitioning function, the agent as a vertex simply processes the data stream locally using the CBT. For a partitioning function like `TopKCount` in which the key-value pairs are shuffled and gradually truncated, we build a single aggregation tree, e.g., Figure 6 middle shows how the *groupby*, *aggregate*, *sort* functions are applied for each level-*i* subtree's root for the micro-promotion job. For partitioning functions like `WordCount`, we build *m* aggregation trees to divide the keys into *m* ranges, where each tree is responsible for the reduce function of one range, thus avoiding the root overload when aggregating a large key space. Figure 6 right shows how the '*fat-tree*'-like aggregation tree is built for the product-bundling job.

Cycles. Naiad [20] uses timestamp vectors to realize dataflow cycles, whereas ELF employs multicast services operating on a job's aggregation tree to create feedback loops in which the results obtained for a job's last epoch are re-injected into its sources. Each job's master has complete control over the contents of feedback messages and how often they are sent. Feedback messages, therefore, can be used to go beyond supporting cyclic jobs to also exert application-specific controls, e.g., set a new threshold, synchronize a new batch, install new job functionality for agents to use, etc.

Why SRTs? The use of DHTs affords the efficient construction of aggregation trees and multicast services, as their converging properties guarantee aggregation or multicast to be fulfilled within only $O(logN)$ hops. Further, a single overlay can support many different independent groups, so that the overheads of maintaining a proximity-aware overlay network can be amortized over all those group spanning trees. Finally, because all of these trees share the same set of underlying agents, each agent can be an input leaf, an internal node, the root, or any combination of the above, causing the computation load well balanced. This is why we term these structures "shared reducer trees" (SRTs).

Implementing feedback loops using DHT-based multicast benefits load and bandwidth usage: each message is replicated in the internal nodes of the tree, at each level, so that only m copies are sent to each internal node's m children, rather than having the tree root broadcast N copies to N total nodes. Similarly, coordination across jobs via the DHT's routing methods is entirely decentralized, benefiting scalability and flexibility, the latter because concurrent ELF jobs use event-based methods to remain responsive to other jobs and/or to user interaction.

2.4 ELF API

Subscribe(Id appid)
Vertex sends JOIN message to construct SRT with the root's nodeid equals to appid.

OnTimer()
Callback. Invoked periodically. This handler has no return value. The master uses it for its periodic activities.

SendTo(Id nodeid, PAO paos)
Vertex sends the key-value pairs to the parent vertex with nodeid, resulting in a corresponding invocation of **OnRecv**.

OnRecv(Id nodeid, PAO paos)
Callback. Invoked when vertex receives serialized key-value pairs from the child vertex with nodeid.

Multicast(Id appid, Message message)
Application's master publishes control messages to vertices, e.g., synchronizing CBTs to be emptied; application's master publishes last epoch's result, encapsulated into a message, to all vertices for iterative loops; or application's master publishes new functions, encapsulated into a message, to all vertices for updating functions.

OnMulticast(Id appid, Message message)
Callback. Invoked when vertex receives the multicast message from application's master.

Table 1: Data plane API

Route(Id appid, Message message)
Vertex or master sends a message to another application. The appid is the hash value of the target application's name concatenated with its creator's name.

Deliver(Id appid, Message message)
Callback. Invoked when the application's master receives an outsider message from another application with appid. This outsider message is usually a query for the application's status such as results.

Table 2: Control plane API

Table 1 and Table 2 show the ELF's data and control plane APIs, respectively. The data plane APIs concern data processing within a single application. The control plane APIs are for coordination between different applications.

```
ArrayList<String> topk;
void OnTimer () {
if (this.isRoot()) {
   this.Multicast(hash("micro-promotion"), new topk(topk));
   this.Multicast(hash("micro-promotion"), new update()); }
   }
void OnMulticast(Id appid, Message message) {
if (message instanceof topk) {
   for(String product: message.topk) {
      if(this.hasProduct(product))
      //if it is an topk message, appear discount ... }
      }
//if it is an update message, start a new batch
else if (message instanceof update) {
   //if leaves, flush CBT and update to the parent vertex
   if (!this.containsChild(appid)) {
      PAO paos = cbt.get(appid).flush();
      this.SendTo (this.getParent(appid), paos);
      cbt.get(appid).empty(); }
      }
   }
```

Figure 7: ELF implementation of micro-promotion application

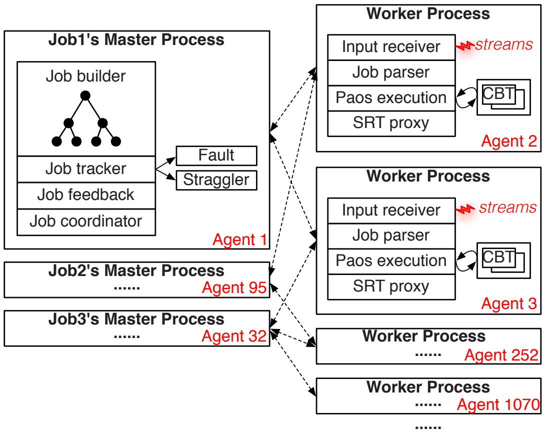


Figure 8: Components of ELF.

A sample use shown in Figure 7 contains partial code for the micro-promotion application. It multicasts update messages periodically to empty agents' CBTs for synchronous batch processing. It multicasts top-k results periodically to agents. Upon receiving the results, each agent checks if it has the top-k product, and if true, the extra discount will appear on the web page. To implement the product-bundling application, the agents subscribe to multiple SRTs to separately aggregate key-value pairs, and agents' associated CBTs are flushed only (without being synchronously emptied), to send a sliding window value to the parent vertices for asynchronously processing. To implement the sale-predication application, the master encapsulates its query and routes to the other two applications to get their intermediate results using `Route`.

3 Implementation

This section describes ELF's architecture and prototype implementation, including its methods for dealing with faults and with stragglers.

3.1 System Architecture

Figure 8 shows ELF's architecture. We see that unlike other streaming systems with static assignments of nodes to act as masters vs. workers, all ELF agents are treated equally. They are structured into a P2P overlay, in which each agent has a unique nodeId in the same flat circular 128-*bit* node identifier space. After an application is launched, agents that have target streams required by the application are automatically assembled into a shared reducer tree (SRT) via their routing substrates. It is only at that point that ELF assigns one or more of the following roles to each participating agent:

Job master is SRT's root, which tracks its own job's execution and coordinates with other jobs' masters. It has four components:

- *Job builder* constructs the SRT to roll up and aggregate the distributed PAOs snapshots processed by local CBTs.

- *Job tracker* detects *key-value* errors, recovers from faults, and mitigates stragglers.
- *Job feedback* is continuously sent to agents for iterative loops, including last epoch's results to be iterated over, new job functions for agents to be updated on-the-fly, application-specific control messages like '*new discount*', etc.
- *Job coordinator* dynamically interacts with other jobs to carry out interactive queries.

Job worker uses a local CBT to implement some application-specific execution model, e.g., asynchronous stream processing with a sliding window, synchronous incremental batch processing with historical records, etc. For consistency, job workers are synchronized by the job master to 'roll up' the intermediate results to the SRT for global aggregation. Each worker has five components:

- *Input receiver* observes streams. Its current implementation assumes logs are collected with Flume [1], so it employs an interceptor to copy stream events, then parses each event into a job-specified *key-value* pair. A typical Flume event is a tuple with *timestamp*, *source IP*, and *event body* that can be split into columns based on different key-based attributes.
- *Job parser* converts a job's SQL description into a workflow of operator functions `f`, e.g., aggregations, grouping, and filters.
- *PAOs execution*: each *key-value* pair is represented as a partial aggregation object (PAO) [9]. New PAOs are inserted into and accumulated in the CBT. When the CBT is "*flushed*", new and past PAOs are aggregated and returned, e.g., $\langle argu, 2, \mathtt{f} : count() \rangle$ merges with $\langle argu, 5, \mathtt{f} : count() \rangle$ to be a PAO $\langle agru, 7, \mathtt{f} : count() \rangle$.
- *CBT* resides in local agent's memory, but can be externalized to SSD or disk, if desired.
- *SRT proxy* is analogous to a socket, to join the P2P overlay and link with other SRT proxies to construct each job's SRT.

A key difference to other streaming systems is that ELF seeks to obtain scalability by changing the system architecture from $1 : n$ to $m : n$, where each job has its own master and appropriate set of workers, all of which are mapped to a shared set of agents. With many jobs, therefore, an agent act as one job's master and another job's worker, or any combination thereof. Further, using DHTs, jobs' reducing paths are constructed with few overlaps, resulting in ELF's management being fully decentralized and load balanced. The outcome is straightforward scaling to large numbers of concurrently running jobs, with each master controls its own job's execution, including to react to failures, mitigate stragglers, alter a job as it is running, and coordinate with other jobs at runtime.

3.2 Consistency

The consistency of states across nodes is an issue in streaming systems that eagerly process incoming records. For instance, in a system counting page views from male users on one node and females on another, if one of the nodes is backlogged, the ratio of their counts will be wrong [28]. Some systems, like Borealis [6], synchronize nodes to avoid this problem, while others, like Storm [4], ignore it.

ELF's consistency semantics are straightforward, leveraging the fact that each CBT's intermediate results (PAOs snapshots) are uniquely named for different timestamped intervals. Like a software combining tree barrier, each leaf uploads the first interval's snapshot to its parent. If the parent discovers that it is the last one in its direct list of children to do so, it continues up the tree by aggregating the first interval's snapshots from all branches, else it blocks. Figure 9 shows an example in which $agent_1$ loses $snapshot_0$, and thus blocks $agent_5$ and $agent_7$. Proceeding in this fashion, a late-coming snapshot eventually blocks the entire upstream path to the root. All snapshots from distributed CBTs are thus sequentially aggregated.

3.3 Fault Recovery

ELF handles transmission faults and agent failures, transient or permanent. For the former, the current implementation uses a simple XOR protocol to detect the integrity of records transferred between each source and destination agent. Upon an XOR error, records will be resent. We deal with agent failures by leveraging CBTs and the robust nature of the P2P overlay. Upon an agent's failure, the dataset cached in the agent's CBT is re-issued, the SRT is re-constructed, and all PAOs are recomputed using a new SRT from the last checkpoint.

In an ongoing implementation, we also use hot replication to support live recovery. Here, each agent in the overlay maintains a routing table, a neighborhood set, and a leaf set. The neighborhood set contains the nodeIds hashed from the webservers' IP addresses that are closest to the local agent. The job master periodically checkpoints each agent's snapshots in the CBT, by asynchronously replicating them to the agent's neighbors.

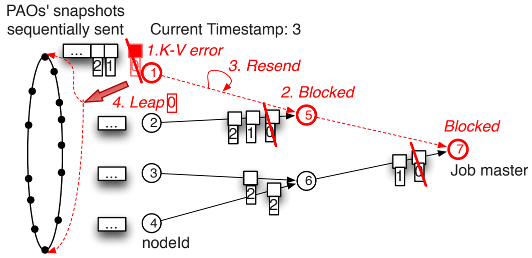


Figure 9: Example of the *leaping straggler* approach. $Agent_1$ notifies all members to discard $snapshot_0$.

ELF's approach to failure handling is similar to that of Spark and other streaming systems, but has potential advantages in data locality because the neighborhood set maintains geographically close (i.e., within the rack) agents, which in turn can reduce synchronization overheads and speed up the rebuild process, particularly in datacenter networks with limited bi-section bandwidths.

3.4 Straggler Mitigation

Straggler mitigation, including to deal with transient slowdown, is important for maintaining low end-to-end delays for time-critical streaming jobs. Users can instruct ELF jobs to exercise two possible mitigation options. First, as in other stream processing systems, speculative backup copies of slow tasks could be run in neighboring agents, termed the "*mirroring straggler*" option. The second option in actual current use by ELF is the "*leaping straggler*" approach, which skips the delayed snapshot and simply jumps to the next interval to continue the stream computation.

Straggler mitigation is enabled by the fact that each agent's CBT states are periodically checkpointed, with a timestamp at every interval. When a CBT's Paos snapshots are rolled up from leaves to root, the straggler will cause all of its all upstream agents to be blocked. In the example shown in Figure 9, $agent_1$ has a transient failure and fails to resend the first checkpoint's data for some short duration, blocking the computations in $agent_5$ and $agent_7$. Using a simple threshold to identify it as a straggler – whenever its parent determines it to have fallen two intervals behind its siblings – $agent_1$ is marked as a straggler. $Agent_5$, can use the *leaping straggler* approach: it invalidates the first interval's checkpoints on all agents via multicast, and then jumping to the second interval.

The *leaping straggler* approach leverages the streaming nature of ELF, maintaining timeliness at reduced levels of result accuracy. This is critical for streaming jobs operating on realtime data, as when reacting quickly to changes in web user behavior or when dealing with realtime sensor inputs, e.g., indicating time-critical business decisions or analyzing weather changes, stock ups and downs, etc.

4 Evaluation

ELF is evaluated with an online social network (OSN) monitoring application and with the well-known `WordCount` benchmark application. Experimental evaluations answer the following questions:

- What performance and functionality benefits does ELF provide for realistic streaming applications (Sec.4.1)?
- What is the throughput and processing latency seen for ELF jobs, and how does ELF scale with number of nodes and number of concurrent jobs (Sec.4.2)?
- What is the overheads of ELF in terms of CPU, memory, and network load (Sec.4.3)?

4.1 Testbed and Application Scenarios

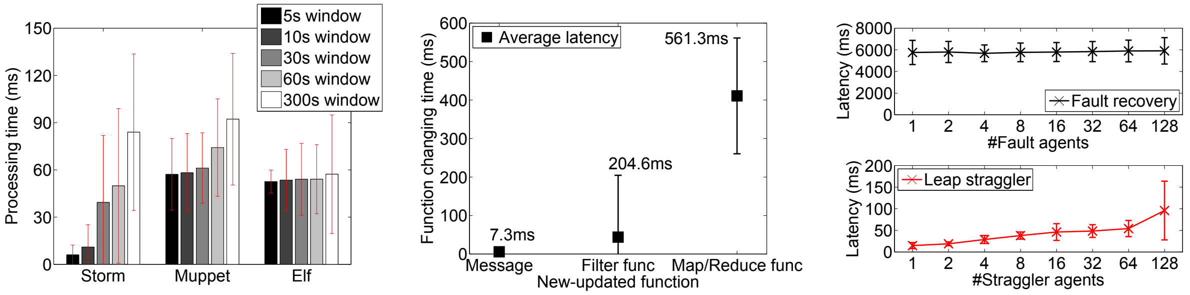
Experiments are conducted on a testbed of 1280 agents hosted by 60 server blades running Linux 2.6.32, all connected via Gigabit Ethernet. Each server has 12 cores (two 2.66GHz six-core Intel X5650 processors), 48GB of DDR3 RAM, and one 1TB SATA disk.

ELF's functionality is evaluated by running an actual application requiring both batch and stream processing. The application's purpose is to identify social spam campaigns, such as compromised or fake OSN accounts used by malicious entities to execute spam and spread malware [13]. We use the most straightforward approach to identify them – by clustering all spam containing the same label, such as an URL or account, into a campaign. The application consumes the events, all labeled as "sales", from the replay of a previously captured data stream from Twitter's public API [3], to determine the top-*k* most frequently twittering users publishing "sales". After listing them as suspects, job functions are changed dynamically, to investigate further, by setting different filtering conditions and zooming in to different attributes, e.g., locations or number of followers.

The application is implemented to obtain online results from live data streams via ELF's agents, while in addition, also obtaining offline results via a Hadoop/HBase backend. Having both live and historical data is crucial for understanding the accuracy and relevance of online results, e.g., to debug or improve the online code. ELF makes it straightforward to mix online with offline processing, as it operates in ways that bypass the storage tier used by Hadoop/HBase.

Specifically, live data streams flow from webservers to ELF and to HBase. For the web tier, there are 1280 emulated webservers generating Twitter streams at a rate of 50 *events/s* each. Those streams are directly intercepted by ELF's 1280 agents, that filter tuples for processing, and concurrently, unchanged streams are gathered by Flume to be moved to the HBase store. The storage tier has 20 servers, in which the name node and job tracker run on a master server, and the data node and task trackers run on the remaining machines. Task trackers are configured to use two map and two reduce slots per worker node. HBase coprocessor, which is analogous to Google's BigTable coprocessor, is used for offline batch processing.

Comparison with Muppet and Storm. With ad-hoc queries sent from a shell via ZeroMQ [5], comparative results are obtained for ELF, Muppet, and Storm, with varying sizes of sliding windows. Figure 10a shows that ELF consistently outperforms Muppet. It achieves performance superior to Storm for large window sizes, e.g., 300 *s*, because the CBT's data structure stabilizes '*flush*'



(a) Comparison of processing times.

(b) Latency for job function changes.

(c) Fault and straggler recovery.

Figure 10: Processing times, latency of function changes, and recovery results of ELF on the Twitter application.

cost by organizing the compressed historical records in an (a,b) tree, enabling fast merging with large numbers of past records.

Job function changes. Novel in ELF is the ability to change job functions at runtime. As Figure 10b shows, it takes less than 10 *ms* for the job master to notify all agents about some change, e.g., to publish discounts in the microsale example. To zoom in on different attributes, the job master can update the filter functions on all agents, which takes less than 210 *ms*, and it takes less than 600 *ms* to update user-specified map/reduce functions.

Fault and Straggler Recovery. Fault and straggler recovery are evaluated with methods that use human intervention. To cause permanent failures, we deliberately remove some working agents from the datacenter to evaluate how fast ELF can recover. The time cost includes recomputing the routing table entries, rebuilding the SRT links, synchronizing CBTs across the network, and resuming the computation. To cause stragglers via transient failures, we deliberately slow down some working agents, by collocating them with other CPU-intensive and bandwidth-aggressive applications. The leap ahead method for straggler mitigation is fast, as it only requires the job master to send a multicast message to notify everyone to drop the intervals in question.

Figure 10c reports recovery times with varying numbers of agents. The top curve shows that the delay for fault recovery is about 7 *s*, with a very small rate of increase with increasing numbers of agents. This is due to the DHT overlay's internally parallel nature of repairing the SRT and routing table. The bottom curve shows that the delay for ELF's leap ahead approach to dealing with stragglers is less than 100 *ms*, because multicast and subsequent skipping time costs are trivial compared to the cost of full recovery.

4.2 Performance

Data streams propagate from ELF's distributed CBTs as leaves, to the SRT for aggregation, until the job master at the SRT's root has the final results. Generated live streams are first consumed by CBTs, and the SRT only picks up truncated *key-value* pairs from CBTs for subsequent shuffling. Therefore, the CBT, as the starting point for parallel streaming computations, directly influences ELF's overall throughput. The SRT, as the tree structure for shuffling *key-value* pairs, directly influences total job processing latency, which is the time from when records are sent to the system to when results incorporating them appear at the root. We first report the per-node throughput of ELF in Figure 11, then report data shuffling times for different operators in Figure 12a 12b. The degree to which loads are balanced, an important ELF property when running a large number of concurrent streaming jobs, is reported in Figure 12c.

Throughput. ELF's high throughput for local aggregation, even with substantial amounts of local state, is based in part on the efficiency of the CBT data structure used for this purpose. Figure 11 compares the aggregation performance and memory consumption of the Compressed Buffer Tree (CBT) with a state-of-the-art concurrent hashtable implementation from Google's `sparsehash` [2]. The experiment uses a microbenchmark running the `WordCount` application on a set of input files containing varying numbers of unique keys. We measure the per-unique-key memory consumption and throughput of the two data structures. Results show that the CBT consumes significantly less memory per key, while yielding similar throughput compared to the hashtable. Tests are run with equal numbers of CPUs (12 cores), and hashtable

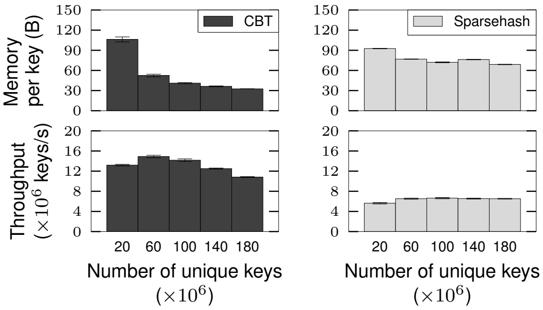
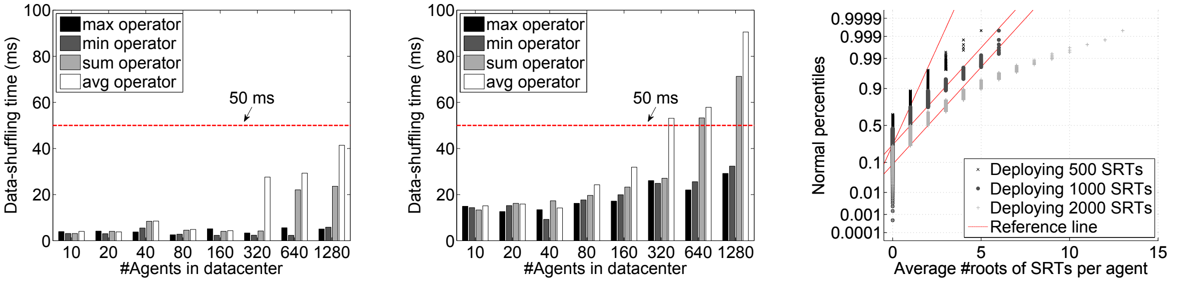


Figure 11: Comparison of CBT with Google Sparsehash.

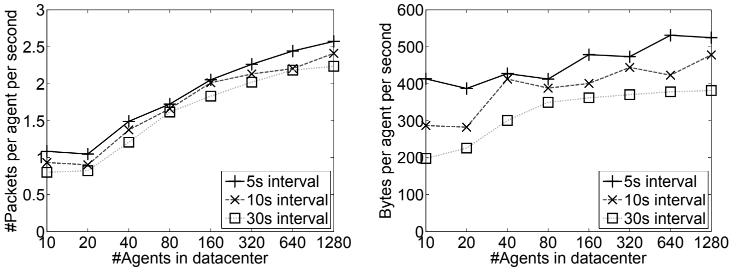


(a) Running operators separately. (b) Running operators simultaneously. (c) Normal probability plot of #roots.

Figure 12: Performance evaluation of ELF on data-shuffling time and load balance.

SPS	CPU	Memory	I/O	C-switch
	%used	%used	wtps	cswsh/s
ELF	**2.96%**	**5.73%**	3.39	780.44
Flume	0.14%	5.48%	2.84	259.23
S-master	0.06%	9.63%	2.96	652.22
S-worker	1.17%	15.91%	11.47	11198.96

SPS: stream processing system.
wtps: write transactions per second.
cswsh/s: context switches per second.



(a) Runtime overheads of ELF vs. others. (b) Additional #packets overhead. (c) Additional bytes overhead.

Figure 13: Overheads evaluation of ELF on runtime cost and network cost.

performance scales linearly with the number of cores.

ELF's per-node throughput of over 1000,000 *keys/s* is in a range similar to Spark Streaming's reported best throughput (640,000 *records/s*) for `Grep`, `WordCount`, and `TopKCount` when running on 4-core nodes. It is also comparable to the speeds reported for commercial single-node streaming systems, e.g., Oracle CEP reports a throughput of 1 million *records/s* on a 16-core server and StreamBase reports 245,000 *records/s* on a 8-core server.

Operators. Figure 12a 12b reports ELF's data shuffling time of ELF when running four operators separately vs. simultaneously. By data shuffling time, we mean the time from when the SRT fetches a CBT's snapshot to the result incorporating it appears in the root. `max` sorts *key-value* pairs in a descending order of value, and `min` sorts in an ascending order. `sum` is similar to `WordCount`, and `avg` refers to the frequency of words divided by the occurrence of *key-value* pairs. As `sum` does not truncate *key-value* pairs like `max` or `min`, and `avg` is based on `sum`, naturally, `sum` and `avg` take more time than `max` and `min`.

Figure 12a 12b demonstrates low performance interference between concurrent operators, because both data shuffling times seen for separate operators and concurrent operators are less than 100 *ms*. Given the fact that concurrent jobs reuse operators if processing logic is duplicated, the interference between concurrent jobs is also low. Finally, these results also demonstrate that SRT scales well with the datacenter size, i.e., number of webservers, as the reduce times increase only linearly with exponential increase in the number of agents. This is because reduce times are strictly governed by an SRT's depth $O(log_{16}N)$, where N is the number of agents in the datacenter.

Balanced Load. Figure 12c shows the normal probability plot for the expected number of roots per agent. These results illustrate a good load balance among participating agents when running a large number of concurrent jobs. Specifically, assuming the root is the agent with the highest load, results show that 99.5% of the agents are the roots of less than 3 trees when there are 500 SRTs total; 99.5% of the agents are the roots of less than 5 trees when there are 1000 SRTs total; and 95% of the agents are the roots of less than 5 trees when there are 2000 SRTs total. This is because of the independent nature of the trees' root IDs that are mapped to specific locations in the overlay.

4.3 Overheads

We evaluate ELF's basic runtime overheads, particularly those pertaining to its CBT and SRT abstractions, and compare them with Flume and Storm. The CBT requires additional memory for maintaining intermediate results, and the SRT generates additional network traffic to maintain the overlay and its tree structure. Table 13a and Figure 13b 13c present these costs, explained next.

Runtime overheads. Table 13a shows the per-node runtime overheads of ELF, Flume, Storm master, and Storm worker. Experiments are run on 60 nodes, each with 12 cores and 48GB RAM. As Table 13a shows, ELF's runtime overheads is small, comparable to Flume, and much less than that of Storm master and Storm worker. This is because both ELF and Flume use a decentralized architecture that distributes the management load across the datacenter, which is not the case for Storm master. Compared to Flume, which only collects and aggregates streams, ELF offers the additional functionality of providing fast, general stream processing along with per-job managemen mechanisms.

Network overheads. Figure 13b 13c show the additional network traffic imposed by ELF with varying update intervals, when running the Twitter application. We see that the number of packets and number of bytes sent per agent increase only linearly, with an exponential increase in the number of agents, at a rate less than the increase in update frequency (from $1/30$ to $1/5$). This is because most packets are ping-pong messages used for overlay and SRT maintenance (initialization and keep alive), for which any agent pings to a limited set of neighboring agents. We estimate from Figure 13b that when scaling to millions of agents, the additional #package is still bounded to 10.

5 Related Work

Streaming Databases. Early systems for stream processing developed in the database community include Aurora [29], Borealis [6], and STREAM [10]. Here, a query is composed of fixed operators, and a global scheduler decides which tuples and which operators to prioritize in execution based on different policies, e.g., interesting tuple content, QoS values for tuples, etc. SPADE [14] provides a toolkit of built-in operators and a set of adapters, targeting the System S runtime. Unlike SPADE or STREAM that use SQL-style declarative query interfaces, Aurora allows query activity to be interspersed with message processing. Borealis inherits its core stream processing functionality from Aurora.

MapReduce-style Systems. Recent work extends the batch-oriented MapReduce model to support continuous stream processing, using techniques like pipelined parallelism, incremental processing for map and reduce, etc.

MapReduce Online [12] pipelines data between map and reduce operators, by calling reduce with partial data for early results. Nova [22] runs as a workflow manager on top of an unmodified Pig/Hadoop software stack, with data passes in a continuous fashion. Nova claims itself as a tool more suitable for large batched incremental processing than for small data increments. Incoop [11] applies memorization to the results of partial computations, so that subsequent computations can reuse previous results for unchanged inputs. One-Pass Analytics [17] optimizes MapReduce jobs by avoiding expensive I/O blocking operations such as reloading map output.

iMR [19] offers the MapReduce API for continuous log processing, and similar to ELF's agent, mines data locally first, so as to reduce the volume of data crossing the network. CBP [18] and Comet [15] run MapReduce jobs on new data every few minutes for "bulk incremental processing", with all states stored in on-disk filesystems, thus incurring latencies as high as tens of seconds. Spark Streaming [28] divides input data streams into batches and stores them in memory as RDDs [27]. By adopting a batch-computation model, it inherits powerful fault tolerance via parallel recovery, but any dataflow modification, e.g., from pipeline to cyclic, has to be done via the single master, thus introducing overheads avoided by ELF's decentralized approach. For example, it takes Spark Streaming seconds for iterating and performing incremental updates, but milliseconds for ELF.

All of the above systems inherit MapReduce's "single master" infrastructure, in which parallel jobs consist of hundreds of tasks, and each single task is a pipeline of map and reduce operators. The single master node places those tasks, launches those tasks, maybe synchronizes them, and keeps track of their status for fault recovery or straggler mitigation. The approach works well when the number of parallel jobs is small, but does not scale to hundreds of concurrent jobs, particularly when these jobs differ in their execution models and/or require customized management.

Large-scale Streaming Systems. Streaming systems like S4 [21], Storm [4], Flume [1], and Muppet [16] use a message passing model in which a stream computation is structured as a static dataflow graph, and vertices run stateful code to asynchronously process records as they traverse the graph. There are limited optimizations on how past states are stored and how new states are integrated with past data, thus incurring high overheads in memory usage and low throughput when operating over larger time windows. For example, Storm asks users to write codes to implement sliding windows for trend topics, e.g., using `Map<>`, `Hashmap<>` data structure. Muppet uses an in-memory hashtable-like data structure, termed a slate, to store past keys and their associated values. Each key-value entry has an update trigger that is run when new records arrive and aggressively inserts new values to the slate. This creates performance issues when the key space is large or when historical windowsize is large. ELF, instead, structures sets of key-value pairs as compressed buffer trees (CBTs) in memory, and uses lazy aggregation, so as to achieve high memory efficiency and throughput.

Systems using persistent storage to provide full fault-tolerance. MillWheel [7] writes all states contained in vertices to some distributed storage system like BigTable or Spanner. Percolator [23] structures a web indexing computation as triggers that run when new values are written into a distributed key-value store, but does not offer consistency guarantees across nodes. TimeStream [24] runs the continuous, stateful operators in Microsoft's StreamInsight [8] on a cluster, scaling with load swings through repartitioning or reconfiguring sub-DAGs with more or less operators. ELF's CBT resides in a local agent's memory, but can be externalized to SSD or disk, if desired, to also fully support fault-tolerance.

ELF is most akin to Naiad [20], which uses vector timestamps to implement cyclic dataflows and also achieves tens of milliseconds for iterations and incremental updates. We differ from Naiad, which sends only data feedback, in that ELF's application-customized master can send feedback messages that can concern data, job control, and new job functions.

In contrast to all of the systems reviewed above, ELF obtains scalability in terms of the number of concurrent jobs run on incoming data via its fully decentralized "many masters" infrastructure. ELF's jobs can differ in their execution models, yet interact to coordinate their actions and/or build on each others' results.

6 Conclusion

ELF implements a novel decentralized model for stream processing that can simultaneously run hundreds of concurrent jobs, by departing from the common "*one master many workers*" architecture to instead, using a "*many masters many workers*" approach. ELF's innovations go beyond the consequent scalability improvements, to also providing powerful programming abstraction for iterative, batch, and streaming processing, and to offer new functionalities that include support for runtime job function change and for cross-job coordination.

Experimental evaluations demonstrate ELF's scalability to up to a thousand concurrent jobs, high per-node throughput, sub-second job latency, and sub-second ability to adjust the actions of jobs being run.

Future work on ELF will go beyond additional implementation steps, e.g., to enhance SRTs for fast aggregation of non-truncated *key-value* pairs, to further optimize performance and to add robustness by extending and experimenting with additional methods for fault recovery.

References

[1] Flume. `http://flume.apache.org/`, 2013.

[2] Sparsehash. `http://code.google.com/p/sparsehash/`.

[3] Twitter streaming apis. `https://dev.twitter.com/docs/streaming-apis`, 2012.

[4] Storm. `https://github.com/nathanmarz/storm.git`.

[5] Zeromq. `http://zeromq.org/`, 2012.

[6] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[7] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *VLDB*, 2013.

[8] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, and e. Tarnavski. Microsoft cep server and online behavioral targeting. *Proc. VLDB Endow.*, 2(2):1558–1561, Aug. 2009.

[9] H. Amur, W. Richter, D. G. Andersen, M. Kaminsky, K. Schwan, A. Balachandran, and E. Zawadzki. Memory-efficient groupby-aggregate using compressed buffer trees. In *SOCC*. 2013.

[10] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[11] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *SOCC*, pages 7:1–7:14. 2011.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*. 2010.

[13] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *IMC*. 2010.

[14] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. Spade: the system s declarative stream processing engine. In *SIGMOD*, pages 1123–1134. 2008.

[15] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: batched stream processing for data intensive distributed computing. In *SOCC*, pages 63–74. 2010.

[16] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan. Muppet: Mapreduce-style processing of fast data. *Proc. VLDB Endow.*, 5(12):1814–1825, Aug. 2012.

[17] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using mapreduce. In *SIGMOD*, pages 985–996. 2011.

[18] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful bulk processing for incremental analytics. In *SOCC*, pages 51–62. 2010.

[19] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum. In-situ mapreduce for log processing. In *USENIXATC*. 2011.

[20] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*. 2013.

[21] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *ICDMW*, pages 170–177. 2010.

[22] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: continuous pig/hadoop workflows. In *SIGMOD*, pages 1081–1090. 2011.

[23] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*. 2010.

[24] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Timestream: reliable stream computation in the cloud. In *Eurosys*, pages 1–14. 2013.

[25] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350. 2001.

[26] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[27] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. 2012.

[28] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*. ACM, Sept. 2013.

[29] S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. The aurora and medusa projects. *Data Engineering*, 51:3, 2003.

Exploiting bounded staleness to speed up Big Data analytics

Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar
Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, Eric P. Xing*
*Carnegie Mellon University, *Intel Labs*

Abstract

Many modern machine learning (ML) algorithms are iterative, converging on a final solution via many iterations over the input data. This paper explores approaches to exploiting these algorithms' convergent nature to improve performance, by allowing parallel and distributed threads to use loose consistency models for shared algorithm state. Specifically, we focus on *bounded staleness*, in which each thread can see a view of the current intermediate solution that may be a limited number of iterations out-of-date. Allowing staleness reduces communication costs (batched updates and cached reads) and synchronization (less waiting for locks or straggling threads). One approach is to increase the number of iterations between barriers in the oft-used Bulk Synchronous Parallel (BSP) model of parallelizing, which mitigates these costs when all threads proceed at the same speed. A more flexible approach, called Stale Synchronous Parallel (SSP), avoids barriers and allows threads to be a bounded number of iterations ahead of the current slowest thread. Extensive experiments with ML algorithms for topic modeling, collaborative filtering, and PageRank show that both approaches significantly increase convergence speeds, behaving similarly when there are no stragglers, but SSP outperforms BSP in the presence of stragglers.

1 Introduction

Large-scale machine learning (ML) has become a critical building block for many applications and services, as the Big Data concept gains more and more momentum. Parallel ML implementations executed on clusters of servers are increasingly common, given the total computation work often involved. These implementations face the same challenges as any parallel computing activity, including performance overheads induced by inter-thread communication and by synchronization among threads.

Among the many ML approaches being used, many fall into a category often referred to as *iterative convergent algorithms*. These algorithms start with some guess at a solution and refine this guess over a number of iterations over the input data, improving a goodness-of-solution objective function until sufficient convergence or goodness has been reached. The key property is convergence, which allows such algorithms to find a good solution given an initial guess. Likewise, minor errors in the adjustments made by any given iteration will not prevent success.

Distributed implementations of iterative convergent algorithms tend to shard the input data and follow the Bulk Synchronous Parallel (BSP) model. The current intermediate solution is shared by all threads, and each worker thread processes its subset of the input data (e.g., news documents or per-user movie ratings). Each worker makes adjustments to the current solution, as it processes each of its input data items, to make it match that item better. In a BSP execution, coordination happens whenever all threads have completed a certain amount of work, which we will refer to as a "clock". All threads work on clock N with a snapshot of the shared state that includes all updates from clock $N-1$, with exchange of updates and a barrier synchronization at the end of each clock.

Although not often discussed as such, BSP relies on the algorithm having a tolerance of staleness. During a given clock, worker threads do not see the adjustments made by others; each of them determines adjustments independently, and those adjustments are aggregated only at the end of the clock. Indeed, these independent adjustments are a source of error that may require extra iterations. But, by coordinating only once per clock, BSP reduces communication costs (by batching updates) and synchronization delays (by reducing their frequency). While most ML practitioners equate an iteration (one pass over the input data) with a BSP clock, doing so fails to recognize staleness as a parameter to be tuned.

This paper describes and analyzes two approaches to more fully exploiting staleness to improve ML convergence speeds. Allowing more staleness often results in faster convergence, but only to a point. While it reduces communication and synchronization, making iterations faster, it can also increase the error in any given iteration. So, there is a tradeoff between decreasing iteration times and increasing iteration counts, determined by the *staleness bound*. In BSP, the maximum staleness corresponds to the work per clock. We find that the best value is often *not* equal to one iteration, and we use the term *Arbitrarily-sized BSP* (A-BSP) to highlight this fact.[1]

A different approach to exploiting staleness is the *Stale Synchronous Parallel* (SSP) model, proposed in our recent workshop paper [12], which generalizes BSP by relaxing the requirement that all threads be working on the same clock at the same time. Instead, threads are allowed to progress a bounded number (the "slack") of clocks ahead of the slowest thread. Like the BSP model, the SSP model bounds staleness (to the product of the slack and

[1] A-BSP is not a different model than BSP, but we use the additional term to distinguish the traditional use of BSP (by ML practitioners) from explicit tuning of staleness in BSP.

the work per clock). But, unlike BSP, SSP's more flexible executions can better mitigate transient straggler effects.

We describe a system, called LazyTable, that supports BSP, A-BSP, and SSP. Using three diverse, real ML applications (topic modeling, collaborative filtering, and PageRank) running on 500 cores, we study the relative merits of these models under various conditions. Our results expose a number of important lessons that must be considered in designing and configuring such systems, some of which conflict with prior work. For example, as expected, we find that tuning the staleness bound significantly reduces convergence times. But, we also find that A-BSP and SSP, when using the (same) best staleness bound, perform quite similarly in the absence of significant straggler effects. In fact, SSP involves some extra communication overheads that can make it slightly slower than A-BSP in such situations. In the presence of transient straggler effects, however, SSP provides much better performance.

This paper makes three primary contributions over previous work, including our workshop paper that proposed SSP. First, it provides the first detailed description of a system that implements SSP, as well as BSP and A-BSP, including techniques used and lessons learned in tuning its performance. Second, to our knowledge, it is the first to introduce the concept of tuning the BSP work-per-clock in the context of parallel ML, allowing A-BSP to be viewed (and evaluated) in the same bounded staleness model as SSP. Third, it provides the first comparative evaluations of A-BSP and SSP, exploring their relative merits when using the same staleness bound, whereas previous papers (e.g., [12]) only compared SSP to BSP. Importantly, these comparisons clarify when SSP does and does not provide value over a simpler A-BSP implementation.

2 Parallel ML and bounded staleness

This section reviews iterative convergent algorithms for ML, the traditional BSP model for parallelizing them, why staleness helps performance but must be bounded, and the A-BSP and SSP approaches to exploiting staleness.

2.1 Iterative convergent algorithms & BSP

Many ML tasks (e.g., topic modeling, collaborative filtering, and PageRank) are mapped onto problems that can be solved via iterative convergent algorithms. Such algorithms typically search a space of potential solutions (e.g., N-dimensional vectors of real numbers) using an *objective function* that evaluates the goodness of a potential solution. The goal is to find a solution with a large (or in the case of minimization, small) objective value. For some algorithms (e.g., eigenvector and shortest path), the objective function is not explicitly defined or evaluated. Rather, they iterate until the solution does not change (significantly) from iteration to iteration.

These algorithms start with an initial state S_0 that has some objective value $f(S_0)$. They proceed through a set of iterations, each one producing a new state S_{n+1} with a potentially improved solution (e.g., greater objective value $f(S_{n+1}) > f(S_n)$). In most ML use-cases, this is done by considering each input datum, one by one, and adjusting the current state to more accurately reflect it. Eventually, the algorithm reaches a stopping condition and outputs the best known state as the solution. A key property of these algorithms is that they will converge to a good state, even if there are minor errors in their intermediate calculations.

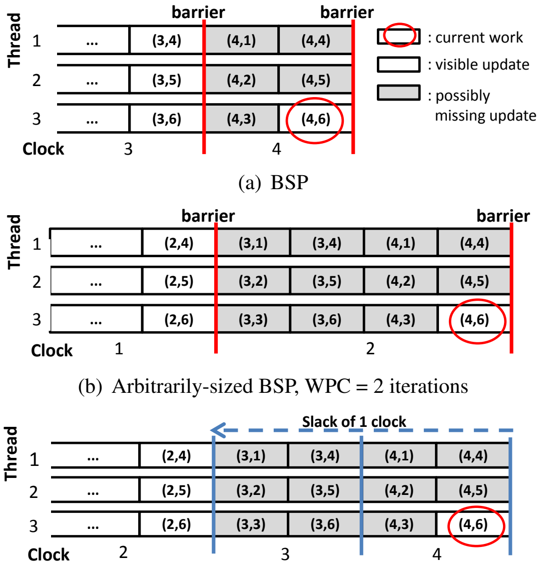
Iterative convergent algorithms are often parallelized with the Bulk Synchronous Parallel (BSP) model. In BSP, a sequence of computation work is divided among multiple computation threads that execute in parallel, and each thread's work is divided into *clock periods* by barriers. The clock period usually corresponds to an amount of work, rather than a wall clock time, and the predominant ML practice is to perform one full iteration over the input data each clock [19]. For an iterative convergent algorithm, the algorithm state is stored in a shared data structure (often distributed among the threads) that all threads update during each iteration. BSP guarantees that all threads see all updates from the previous clock, but not that they will see updates from the current clock, so computation threads can experience *staleness errors* when they access the shared data.

2.2 Bounded staleness

Parallel execution always faces performance challenges due to inter-thread communication overheads and synchronization delays. They can be mitigated by having threads work independently, but at the expense of threads not seeing the latest solution improvements by other threads. This lack of awareness of updates to the shared state by other threads is what we mean by "staleness".

In the BSP model, threads work independently within a clock period, with no guarantee of seeing updates from other threads until the next barrier. Figure 1(a) illustrates a BSP execution with 3 threads. In the original sequential execution, each iteration has 6 *work-units*, which are finer-grained divisions of the original sequential execution. We denote (i, j) as the j-th work-unit of the i-th iteration. In this example, when thread-3 is doing work (4, 6), which is circled in the illustration, BSP only guarantees that it will see the updates from work completed in the previous clocks (clocks 3 and lower, not shaded). It may or may not see updates from the five shaded work-units.

Because iterative convergent algorithms can tolerate some error in the adjustments made to the algorithm state, independent work by threads can be acceptable even though the algorithm state is shared. That is, even if a thread incorrectly assumes that other threads have made



(a) BSP

(b) Arbitrarily-sized BSP, WPC = 2 iterations

(c) SSP, slack = 1 clock (note: no barriers)

Figure 1: BSP, A-BSP, and SSP models. A block with (i, j) represents the j-th work-unit of the i-th iteration. Focusing on thread 3's execution of the circled work-unit, the shaded blocks indicate the updates that it may not see, under each model. SSP is more flexible than A-BSP, allowing the work of later clocks to start before the work of earlier clocks complete, up to the slack bound.

no relevant modifications to the shared state, causing it to produce a somewhat imperfect adjustment, the algorithm will still converge. Exploiting this fact, such as with a BSP implementation, allows parallel execution without synchronizing on every update to the shared state.

Accepting some staleness allows batching of updates, more reading from local (possibly out-of-date) views of the shared state, and less frequent synchronization, all of which helps iterations to complete faster. But, it may take more iterations to converge, because each iteration is less effective. In theory, a number of algorithms have been shown to converge given reasonable bounds on staleness [23]. Empirically, our experiments show that there is a sweet spot in this staleness tradeoff that maximizes overall convergence speed for a given execution, considering both the time-per-iteration and the effectiveness-per-iteration aspects.

Note that having a bound on staleness is important, at least in theory. There have been *Asynchronous Parallel* systems [1] that allow threads to work completely asynchronously, with best-effort communication of updates among them, but their robustness is unknown. While they have worked in some empirical evaluations, the convergence proofs associated with such efforts assume there are bounds on how out-of-synch threads will get, even though such systems (in contrast to those we consider) provide no mechanisms to enforce such bounds.

2.3 Expanding staleness exploitation

This section describes two approaches for more fully exploiting staleness to improve ML convergence speeds.

Arbitrarily-sized Bulk Synchronous Parallel (A-BSP). Because the staleness bound represents a tradeoff, tuning it can be beneficial. Focusing first on the BSP model, we define the amount of work done in each clock period as *work-per-clock (WPC)*. While the traditional ML approach equates iteration and clock, it is not necessary to do so. The WPC could instead be a multiple of or a fraction of an iteration over the input data. To distinguish BSP executions where WPC is not equal to one iteration from the current ML practice, we use the term "Arbitrarily-sized BSP" (A-BSP) in this paper.

Figure 1(b) illustrates an A-BSP execution in which the WPC is two full iterations. That is, the barriers occur every two iterations of work, which approximately halves the communication work and doubles the amount of data staleness compared to base BSP. Manipulating the A-BSP WPC in this manner is a straightforward way of controlling the staleness bound.

Stale Synchronous Parallel (SSP). While A-BSP amortizes per-clock communication work over more computation, it continues to suffer from BSP's primary performance issue: stragglers. All threads must complete a given clock before the next clock can begin, so a single slow thread will cause all threads to wait. This problem grows with the level of parallelism, as random variations in execution times increase the probability that at least one thread will run unusually slowly in a given clock. Even when it is a different straggler in each clock, due to transient effects, the entire application can be slowed significantly (see Section 5 for examples).

Stragglers can occur for a number of reasons including heterogeneity of hardware [21], hardware failures [3], imbalanced data distribution among tasks, garbage collection in high-level languages, and even operating system effects [5, 27]. Additionally, there are sometimes algorithmic reasons to introduce a straggler. Many algorithms use an expensive computation to check a stopping criterion, which they perform on a different one of the machines every so many clocks.

Recently, a model called "Stale Synchronous Parallel" (SSP) [12] was proposed as an approach to mitigate the straggler effect. SSP uses work-per-clock as defined above, but eliminates A-BSP's barriers and instead defines an explicit *slack* parameter for coordinating progress among the threads. The slack specifies how many clocks out-of-date a thread's view of the shared state can be, which implicitly also dictates how far ahead of the slowest thread that any thread is allowed to progress. For example, with a slack of s, a thread at clock t is guaranteed to see all updates from clocks 1 to $t - s - 1$, and it may see (not guaranteed) the updates from clocks $t - s$ to $t - 1$.

Figure 1(c) illustrates an SSP execution with a slack of 1 clock. When thread-3 is doing work (4, 6), SSP guarantees that it sees all the updates from clocks 1 and 2, and it might also see some updates from clocks 3 and 4.

Relationship of A-BSP and SSP. In terms of data staleness, SSP is a generalization of A-BSP (and hence of BSP), because SSP's guarantee with slack set to zero matches A-BSP's guarantee when both use the same WPC. (Hence, SSP's guarantee with slack set to zero and WPC set to 1 iteration matches BSP's guarantee.) For convenience, we use the tuple $\{wpc, s\}$ to denote an SSP or A-BSP configuration with work per clock of wpc and slack of s. (For A-BSP, s is always 0.) The data staleness bound for an SSP execution of $\{wpc, s\}$ is $wpc \times (s+1) - 1$. This SSP configuration provides the same staleness bound as A-BSP with $\{wpc \times (s+1), 0\}$.

A-BSP requires a barrier at the end of each clock, so it is very sensitive to stragglers in the system. SSP is more flexible, in that it allows some slack in the progress of each thread. The fastest thread is allowed to be ahead of the slowest by $wpc \times s$. As a result, the execution of SSP is like a pipelined version of A-BSP, where the work of later clocks can start before the work of earlier clocks complete. Intuitively, this makes SSP better at dealing with stragglers, in particular when threads are transient stragglers that can readily resume full speed once the cause of the slowness mitigates (e.g., the stopping criterion calculation or the OS/runtime operation completes).

But, SSP involves additional communication costs. The SSP execution of $\{wpc, s\}$ will have $s+1$ times more clocks than its A-BSP counter-part. In other words, SSP is a finer-grained division of the execution, and updates will be propagated at a higher frequency. As a result, SSP requires higher network throughput and incurs extra CPU usage for communication. When there are no stragglers, A-BSP can perform slightly better by avoiding the extra communication. Our evaluation explores this tradeoff.

3 LazyTable design and implementation

This section describes LazyTable, which provides for shared global values accessed/updated by multiple threads across multiple machines in a *staleness-aware* manner. It can be configured to support BSP, A-BSP, and SSP.

LazyTable holds globally shared data in a cluster of *tablet servers*, and application programs access the data through the *client library*. (See Figure 2.) Each tablet server holds a partition (shard) of the data, and the client library services data access operations from the application by communicating with the appropriate tablet server. The client library also maintains a hierarchy of caches and operation logs in order to reduce network traffic.

3.1 LazyTable data model and API

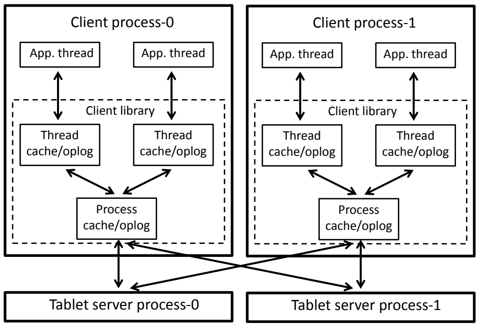


Figure 2: LazyTable running two application processes with two application threads each.

Data model. Globally shared data is organized in a collection of *rows* in LazyTable. A row is a user-defined data type and is usually a container type, such as an STL vector or map. The row type is required to be serializable and be defined with an associative aggregation operation, such as plus, multiply, or union, so that updates from different threads can be applied in any order. Each row is uniquely indexed by a (table_id, row_id) tuple.

Having each data unit be a row simplifies the implementation of the many ML algorithms that are naturally expressed as operations on matrices and vectors. Each vector can naturally be stored as a row in LazyTable.

Operations. LazyTable provides a simple API for accessing the shared data and for application threads to synchronize their progress. Listed below are the core methods of the LazyTable client library, which borrow from Piccolo [28] and add staleness awareness:

`read(table_id, row_id, slack)`: Atomically retrieves a row. Ideally, the row is retrieved from a local cache. If no available version of the row is within the given `slack` bound, the calling thread waits. This is the only function that blocks the calling thread.

`update(table_id, row_id, delta)`: Atomically updates a row by `delta` using the defined aggregation operation. The `delta` should have the same type as row data, so it is usually a vector instead of a single value. If the target row does not exist, the row data will be set to `delta`.

`refresh(table_id, row_id, slack)`: Refreshes the process cache entry of a row, if it is too old. This interface can be used for the purpose of prefetching.

`clock()`: Increases the "clock time" of the calling thread by one. Although this is a synchronization-purposed function, it does not block the calling thread, so it is different from a barrier in a BSP or A-BSP system.

Data freshness and consistency guarantees. LazyTable does not have explicit barriers at the end of each clock. Instead, it bounds staleness by attaching a *data age* field to each row. If a row has a data age of τ, it is guaranteed to contain all updates from all application threads for clocks $1, 2, ..., \tau$. For the case of BSP, where each thread can only use the output from the previous clock, a thread at clock t can only use the data of age $t-1$. For the case of SSP, when a thread at clock t issues a `read` request with a slack of s, only row data with data age $\tau \geq t-1-s$ can be returned.

LazyTable also enforces the *read-my-updates* property, which ensures that the data read by a thread contains all its own updates. Read-my-updates often makes it much easier to reason about program correctness. It also enables applications to store local intermediate data in LazyTable.

3.2 LazyTable system components

The LazyTable prototype is written in C++, using the ZeroMQ [32] socket library for communication.

3.2.1 Tablet servers

The tablet servers collectively store the current "master" view of the row data. Data is distributed among the tablet servers based on row ID, by default using a simple, static row-server mapping: $tablet\_id = row\_id$ mod $num\_of\_tablets$. The current implementation does not replicate row data or support multi-row updates, and also requires the tablet server data to fit in memory.

Each tablet server uses a vector clock to keep track of the version of all its row data. Each entry of the vector clock represents the server's view of the progress of each client process, which starts from zero and is increased when a `clock` message is received. The minimal clock value among the vector clock entries is referred to as the *global clock value*. A global clock value of t indicates that all application threads on all client machines have finished the work up to clock t, and that all updates from these threads have been merged into the master data. The `update` and `read` requests are serviced by the tablet servers as follows:

Proposing updates. When the tablet server receives a row update from a client, it puts it into a *pending updates list*. Updates in this list are applied to the master data only after a `clock` message is received from that client. This mechanism guarantees that the vector clock values can uniquely determine the version of the master data.

Reading values. When the tablet server receives a read request from a client, it looks at its global clock value. If the clock value is at least as large as the requested data age, the request is serviced immediately. Otherwise, the request will be put in the *pending read list*, which is sorted by the requested data age. When the global clock value of the tablet server advances to a required data age, the server then replies to those pending requests in the list. Along with the requested row data, the server also sends two clock fields: data age and *requester clock*. The data age is simply the server's global clock. The requester clock is the server's view of the requesting client's clock—this indicates which updates from that client have been applied to the row data. The client uses this information to clear its oplogs (discussed below).

3.2.2 Client library

The client library runs in the same process as the application code and translates the LazyTable API calls to server messages. It also maintains different levels of *caches* and *operation logs*. The client library creates several *background worker threads* that are responsible for jobs such as propagating updates and receiving data.

The LazyTable client library has two levels of caches/oplogs: the process cache/oplog and the thread cache/oplog, as depicted in Figure 2. The process cache/oplog is shared by all threads in the client process, including the application threads and background worker threads. Each thread cache/oplog, on the other hand, is exclusively associated with one application thread. Thread cache entries avoid locking at the process cache, and they are used only for the few rows that would suffer contention. The use of these caches/oplogs in servicing `update` and `read` operations is described below.

The client library tracks the progress of its application threads using a vector clock. Each entry of the vector clock represents the progress of one application thread, which starts at zero and will be increased by one each time the application thread calls the `clock()` operation.

Proposing updates. Application threads propose updates to the globally shared data using the `update()` operation. Suppose a thread at clock t wants to update the value of a row by `delta`. If the corresponding thread cache/oplog entry exists, the update will be logged in the thread oplog. To guarantee "read-my-updates", it will also be applied to the data in the thread cache immediately. When this thread calls the `clock` function, all updates in the thread oplog will be pushed to the process oplog and also applied to the row data in the process cache. If there is no thread cache/oplog entry, then the update will be pushed to the process cache/oplog immediately.

When all application threads in a client process have finished clock t, the client library will signal a background thread to send the `clock` messages, together with the batched updates of clock t in the process oplog, to the tablet servers. For robustness, the process oplogs are retained until the next time the client receives row data containing that update.

Reading values. When an application thread at clock t wants to read row r with a slack of s, the client library will translate the request to "read row r with data age $age \geq t-s-1$". To service this request, the client library will first look in the thread cache, and then the process cache, for a cached entry that satisfies the data age requirement. If not, it will send a request to the tablet server for row r and block the calling thread to wait for the new data. A per-row tag in the process cache tracks whether a request is in progress, in order to squash redundant requests to the same row.

When the server replies, the row data is received by a background worker thread. The server also sends a requester clock value rc, which tells the client which of its updates have been applied to this row data. The receiver thread erases from its process oplog any operation for that tablet server's shard with clock $\leq rc$. Then, to guarantee "read-my-updates" for the application thread, it applies to the received data row (r) any operations for row r remaining in the process oplog (i.e., with clock $> rc$). Finally, the receiver thread sets the data age field of the row and signals the waiting application threads.

3.3 Prefetching and fault-tolerance

Prefetching. Not unexpectedly, we find that prefetching makes a huge difference in performance, even when bounded staleness is allowed (e.g., see Section 5.5). But, the access patterns often do not conform to traditional application-unaware policies, such as sequential prefetching. So, LazyTable provides an explicit `refresh()` interface for prefetching. The parameters for `refresh()` are the same as `read()`, but the calling thread is not blocked and row data is not returned.

Applications usually prefetch row data at the beginning of each clock period. The general rule-of-thumb is to refresh every row that will be used in that clock period so as to overlap fetch time with computation. While this requires an application to know beforehand what it will read in the current clock, many iterative ML applications have this property. Generally, each application thread processes the same input data in the same order, accessing the same rows in the same order as well. To leverage this property, LazyTable provides an automatic prefetcher module. The access pattern can be captured after one iteration, and the prefetcher can automatically refresh the needed data at the beginning of each clock.

The prefetching described so far addresses read miss latencies. But, for SSP, where multiple versions of the data can be accepted by a `read`, prefetching can also be used to provide fresher data. LazyTable supports two prefetching strategies. `Conservative prefetching` only refreshes when necessary; if $cache\_age < t - s - 1$, the prefetcher will send a request for (row $= r$, age $\geq t - s - 1$). `Aggressive prefetching` will always refresh if the row is not from the most recent clock, seeking the freshest possible value.

The conservative prefetching strategy incurs the minimal amount of traffic in order to avoid freshness misses. The aggressive prefetching strategy refreshes the data frequently, even with an infinite slack, at the cost of extra client-server communication. As a result, we can use infinite slack plus aggressive prefetching to emulate Asynchronous Parallel systems (as discussed in Section 2.2) in LazyTable. In our experiments to date, it is usually worthwhile to pay the cost of aggressive prefetching, so we use it as the default prefetching strategy.

Fault tolerance. LazyTable provides fault-tolerance via checkpointing. The tablet servers are told (in advance) about the clock at which to make a checkpoint, so that they can do so independently. Suppose the tablet servers are planned to make a checkpoint at the end of clock t. One way of checkpointing is to have each tablet server flush all its master data to the storage (take a snapshot) as soon as it has received a `clock`=t message from all of the clients. But, when slack > 0, the snapshot taken from this approach is not a *pure* one that contains exactly the clock 1 through t updates from all clients: some clients may have run ahead and already applied their clock $t + 1, \ldots$ updates to the master data. We call this approach an *approximate snapshot*. Approximate snapshots can be used to do off-line data processing after the computation, such as expensive objective value computation. LazyTable currently implements only approximate snapshot.

An alternative is to have each tablet server keep a *pure copy* of the master data, and have the tablet server flush it to storage instead of the latest master data. If a tablet server is going to checkpoint at the end of clock t, we can keep all updates beyond clock t out of the pure copy, so that it contains exactly the updates from clock 1 through t. But, some effects from SSP slack can be present. For example, client-1 might have generated its updates for clock t based on client-2's updates from clock $t + 1$. When one restores the execution from clock t using that snapshot, there will still be a residual influence from the "future". Moreover, in the view of client-1, client-2 goes backwards. Fortunately, iterative convergent algorithms suitable for SSP can tolerate this kind of error as well [18].

4 Example ML applications

This section describes three ML apps with different algorithms, representing a range of ML approaches.

Topic modeling. Topic modeling is a class of problems that assign *topic vectors* to documents. The specific model on which we focus is known as Latent Dirichlet Allocation [6] (LDA). LDA learns the parameters of the document-topic and topic-word distributions that best explain the input data (a corpus of documents). While there are a number of ways to do this, our example application uses the Gibbs sampling [17] algorithm.

Our implementation works as follows. We divide the set of input documents among threads and use two tables to store document-topic assignment and word-topic respectively. In each iteration, each thread passes through the words in their input documents. For the word $word\_id$ in document doc_id, the thread reads the doc_id-th row of the document-topic table and the $word\_id$-th row plus a summation row[2] of the word-topic table, and updates

[2] The sum of all word-topic rows, used by the algorithm to calculate

them based on the calculated topic assignment.

Matrix factorization. Matrix factorization can be used to predict missing values in a sparse matrix.[3] One example application is to predict user's preferences based on the known preferences of similar users, where the sparse matrix represents users' preference ranking to items. Matrix factorization assumes the matrix has low rank and can be expressed as the product of a tall skinny matrix (the *left-matrix*) and a short wide matrix (the *right-matrix*). Once this factorization is found, any missing value can be predicted by computing the product of these two matrices. This problem can be solved by the stochastic gradient descent algorithm [14].

Our implementation on LazyTable partitions the known elements in the sparse matrix among threads and uses two tables to store the left-matrix and right-matrix, respectively. In each iteration, each thread passes through the elements, and for the element (i, j) in the sparse matrix, it reads and adjusts the i-th row of the left-matrix table and the j-th row of the right-matrix table.

PageRank. The PageRank algorithm assigns a weighted score (PageRank) to every vertex in a graph [10], the score of a vertex measures its importance in the graph.

We implement an edge-scheduling version of PageRank on LazyTable. In each iteration, the algorithm passes through all the edges in the graph and updates the rank of the dst node according to the rank of the src node. The set of edges are partitioned evenly among threads, and the application stores the ranks of each node in LazyTable.

5 Evaluation

This section evaluates the A-BSP and SSP approaches via experiments with real ML applications on our LazyTable prototype; using the same system for all experiments enables us to focus on these models with all else being equal. The results support a number of important findings, some of which depart from previous understandings, including: (1) the staleness bound controls a tradeoff between iteration speed and iteration goodness, in both A-BSP and SSP, with the best setting generally being greater than a single iteration; (2) SSP is a better approach when transient straggler issues occur; (3) SSP requires higher communication throughput, including CPU usage on communication; and (4) iteration-aware prefetching significantly lowers delays for reads and also has the effect of reducing the best-choice staleness bound.

5.1 Evaluation setup

Cluster and LazyTable configuration. Except where otherwise stated, the experiments use 8 nodes of the NSF PRObE cluster [15]. Each node has four 2.1 GHz 16-core Opteron 6272s and 128GB of RAM (512 cores in total). The nodes run Ubuntu 12.04 and are connected via an Infiniband network interface (40Gbps spec; ≈13Gbps observed).

A small subset of experiments, identified explicitly when described, use a second cluster (the "VM cluster"). It consists of 32 8-core blade servers running VMware ESX and connected by 10 Gbps Ethernet. We create one VM on each physical machine, configured with 8 cores (either 2.3GHz or 2.5GHz each) and 23GB of RAM, running Debian Linux 7.0.

Each node executes a client process, with one application thread per core, and a tablet server process. The default aggressive prefetching policy is used, unless otherwise noted. The staleness bound configuration for any execution is described by the "wpc" and "slack" values. The units for the WPC value is iterations, so wpc=2 means that two iterations are performed in each clock period. The units for slack is clocks, which is the same as iterations if wpc=1. Recall that, generally speaking, BSP is wpc=1 and slack=0, A-BSP is wpc=N and slack=0, and SSP is wpc=1 and slack=N.

Application benchmarks. We use the three example applications described in Section 4: Topic Modeling (TM), Matrix Factorization (MF), and PageRank (PR). Table 1 summarizes the problem sizes.

App.	# of rows	Row size (bytes)	# of row accesses per iteration
TM	400k	800	600m
MF	500k	800	400m
PR	685k	24	15m

Table 1: Problem sizes of the ML application benchmarks.

For TM, we use the *Nytimes* dataset, which contains 300k documents, 100m words, and a vocabulary size of 100k. We configure the application to generate 100 topics on this dataset. The quality of the result is defined as the loglikelihood of the model, which is a value that quantifies how likely the model can generate the observed data. A higher value indicates a better model.

For MF, we use the *Netflix* dataset, which is a 480k-by-18k sparse matrix with 100m known elements. We configure the application to factor it into the product of two matrices with rank 100, using an initial step size of 5e-10. Result quality is defined as the summation of squared errors, and a lower value indicates a better solution.

For PageRank, we use the *web-BerkStan* dataset [24], which is a web graph with 685k nodes and 7.6m edges. We configure the application to use a damping factor of 0.85. Because there is no result quality criteria for PageRank, we define it to be the summation of squared errors from a "ground truth result", which we obtain by running a sequential PageRank algorithm on a single machine for a relatively large number of iterations (100).

the probability that a word belongs to a certain topic.

[3] Here "sparse" means most elements are unknown.

A low value indicates a better solution.

We extract result quality data 16 times, evenly spaced, during each execution. To do so, the application creates a background thread in each client process and rotates the extraction among them. For TM and PR, the result quality is computed during the execution by a background thread in each client. For MF, the computation is very time-consuming (several minutes), so we instead have LazyTable take snapshots (see Section 3.3) and compute the quality off-line.

5.2 Exploiting staleness w/ A-BSP and SSP

A-BSP. Figure 3(a) shows performance effects on TM of using A-BSP with different WPC settings. The leftmost graph shows overall *convergence speed*, which is result quality as a function of execution time, as the application converges. WPC settings of 2 or 4 iterations significantly outperform settings of 1 (BSP) or 8, illustrating the fact that both too little and too much staleness is undesirable. One way to look at the data is to draw a horizontal line and compare the time (X axis) required for each setting to reach a given loglikelihood value (Y axis). For example, to reach -9.5e8, WPC=1 takes 236.2 seconds, while WPC=2 takes only 206.6 seconds.

The middle and rightmost graphs help explain this behavior, showing the *iteration speed* and *iteration effectiveness*, respectively. The middle graph shows that, as WPC increases, iterations complete faster. The rightmost graph shows that, as WPC increases, each iteration is less effective, contributing less to overall convergence such that more iterations are needed. The overall convergence speed can be thought of as the combination of these two metrics. Because the iteration speed benefit exhibits diminishing returns, and the iteration effectiveness does not seem to, there ends up being a sweet spot.

SSP. Figure 3(b) shows the same performance effects when using SSP with different slack settings. Similar trends are visible: more staleness increases iteration speed, with diminishing returns, and decreases iteration effectiveness. Empirically, the sweet spot is at slack=3.

We show the behavior for *infinite* slack, for comparison, which is one form of non-blocking asynchronous execution. Although it provides no guarantees, it behaves reasonably well in the early portion of the execution, because there are no major straggler issues and the aggressive prefetching mechanism refreshes the data even though it is not required. But, it struggles in the final (fine-tuning) stages of convergence. Even the BSP baseline (slack=0) finishes converging faster.

A-BSP vs. SSP. Figure 3(c) uses the same three-graph approach to compare four configurations: BSP, the best-performing A-BSP (wpc=4), the best-performing SSP from above (slack=3), and the best-performing overall configuration (a hybrid with wpc=2 and slack=1). Other than BSP, they all use the same staleness bound: wpc × (slack + 1). The results show that SSP outperforms both BSP and A-BSP, and that the best configuration is the hybrid. For example, to reach -9.5e8, A-BSP takes 237.0 seconds, while the two SSP options take 222.7 seconds and 193.7 seconds, respectively. Indeed, across many experiments with all of the applications, we generally find that the best setting is slack=1 with wpc set to one-half of the best staleness bound value. The one clock of slack is enough to mitigate intermittent stragglers, and using larger WPC amortizes communication costs more.

Looking a bit deeper, Figure 4 shows a breakdown of iteration speed into two parts: computation time and "wait time", which is the time that the client application is blocked by a LazyTable `read` due to data freshness misses. As expected, the iteration speed improvements from larger staleness bounds (from the first bar to the second group to the third group) come primarily from reducing wait times. Generally, the wait time is a combination of waiting for stragglers and waiting for fetching fresh-enough data. Because there is minimal straggler behavior in these experiments, almost all of the benefit comes from reducing the frequency of client cache freshness misses. The diminishing returns are also visible, caused by the smaller remaining opportunity (i.e., the wait time) for improvement as the staleness bound grows.

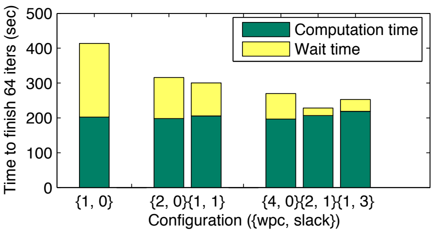
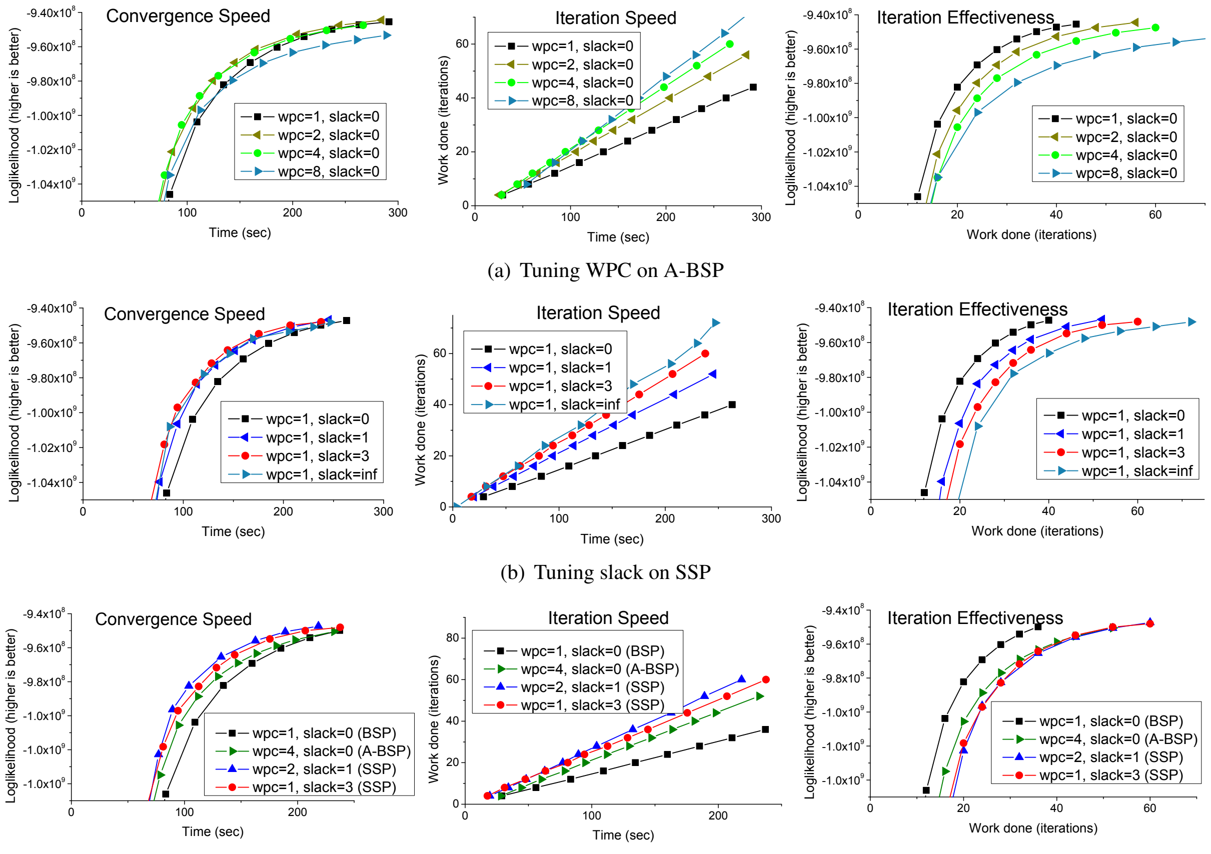


Figure 4: Time consumption distribution of TM.

Matrix factorization and PageRank. Space limits preclude us from including all results, but Figure 5 shows the convergence speed comparison. For MF, we observe less tolerance for staleness, and SSP with slack=1 performs the best, while A-BSP with wpc=2 actually struggles to converge.[4]

PageRank behaves more similarly to Topic Modeling, but we found that A-BSP $\{4,0\}$ slightly outperforms SSP $\{2,1\}$ and significantly outperforms SSP $\{1,3\}$ in this case. This counter-intuitive result occurs because of the increased communication overheads associated with SSP, which aggressively sends and fetches data every clock, combined with PageRank's lower computation work. When the communication throughput, not syn-

[4] We found, empirically, that MF does not diverge as shown with smaller step sizes, which bound the error from staleness to lower values, but we show this data because it is for the best step size for BSP.



(a) Tuning WPC on A-BSP

(b) Tuning slack on SSP

(c) Direct comparison of SSP with A-BSP

Figure 3: Different ways of exploiting data staleness in Topic Modeling.

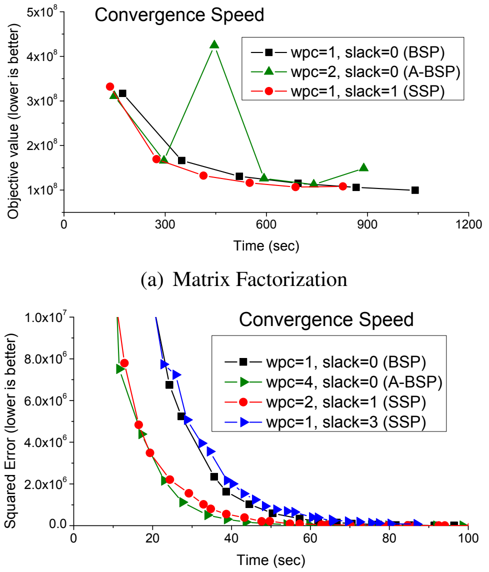
chronization overheads or straggler delays, limit iteration speeds, SSP's tendency to use more communication can hurt performance if it does so too aggressively. Note, however, that maximizing WPC with slack=1 remains close to the best case, and is more robust to larger straggler issues. We will examine this communication overhead of SSP in more detail in Section 5.4.

5.3 Influence of stragglers

The original motivation for SSP was to tolerate intermittent stragglers [12], but our carefully controlled experimental setup exhibits minimal execution speed variation. This section examines Topic Modeling performance under two types of straggler behavior.

Delayed threads. First, we induced stragglers by having application threads sleep in a particular schedule, confirming that a complete system addresses such stragglers as predicted in the HotOS paper [12]. Specifically, the threads on machine-1 sleep d seconds at iteration-1, and then the threads on machine-2 sleep d seconds at iteration-2, and so on. Figure 6 shows the average time per iteration, as a function of d, for Topic Modeling via BSP, A-BSP, and SSP. Ideally, the average time per iteration would increase by only $\frac{d}{N}$ seconds, where N is the number of machines, because each thread is delayed d seconds every N iterations. For A-BSP, the average iteration time increases linearly with a slope of 0.5, because the threads synchronize every two iterations (wpc=2), at which time one delay of d is experienced. For SSP, the effect depends on the magnitude of d relative to the un-delayed time per iteration (≈4.2 seconds). When d is 6 or less, the performance of SSP is close to ideal, because the delays are within the slack. Once d exceeds the amount that can be mitigated by the slack, the slope matches that of A-BSP, but SSP stays 1.5 seconds-per-iteration faster than A-BSP.

Background work. Second, we induced stragglers with competing computation, as might occur with background activities like garbage collection or short high-priority jobs. We did these experiments on 32 machines of the VM cluster, using a background "disrupter" process. The disrupter process creates one thread per core (8 in the VM cluster) that each perform CPU-intensive work when activated; so when the disrupter is active, the CPU scheduler will give it half of the CPU resources (or more, if the TM thread is blocked). Using discretized time slots of size t, each machine's disrupter is active in a time slot with a probability of 10%, independently determined.



(a) Matrix Factorization

(b) PageRank

Figure 5: Synchronization-staleness tradeoff examples.

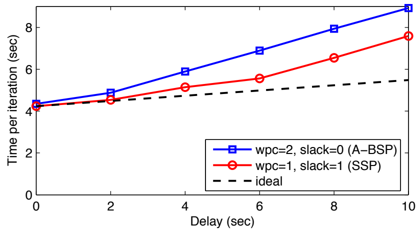


Figure 6: Influence of delayed threads.

Figure 7 shows the increase in iteration time as a function of t. Ideally, the increase would be just 5%, because the disrupters would take away half of the CPU 10% of the time. The results are similar to those from the previous experiment. SSP is close to ideal, for disruptions near or below the iteration time, and consistently able to mitigate the stragglers' impact better than A-BSP.

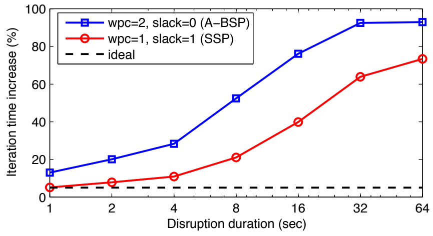


Figure 7: Influence of background disrupting work. With no disruption, each iteration takes about 4.2 seconds.

5.4 Examining communication overhead

Table 2 lists the network traffic of A-BSP {4, 0}, SSP {2, 1}, and SSP {1, 3} for Topic Modeling. We denote the traffic from client to server as "sent" and that from server to client as "received". The "sent" traffic is mainly caused by `update`, and the "received" traffic is mainly caused by `read`. For a configuration that halves the WPC, the sent traffic will be doubled, because the number of clocks for the same number of iterations is doubled. However, the received traffic is less than doubled, because for SSP, if the data received from the server is fresh enough, it can be used in more than one clock of computation.

As discussed in Section 5.2, this extra communication can cause SSP to perform worse than A-BSP in some circumstances. Much of the issue is the CPU overhead for processing the communication, rather than network limitations. To illustrate this, we emulate a situation where there is zero computation overhead for communication. Specifically, we configure the application to use just 4 application threads, and then configure the VMs to use either 4 cores (our normal 1 thread/core) or 8 cores (leaving 4 cores for doing the communication processing without competing with the application threads). Because we have fewer application threads, using WPC=1 would be a lot of computation work per clock. As a result, we make WPC smaller so that it is similar to our previous results: {wpc=0.25, slack=0} for A-BSP and {wpc=0.125, slack=1} for SSP. Figure 8 shows the time for them to complete the same amount of work. The primary takeaway is that having the extra cores makes minimal difference for A-BSP, but significantly speeds up SSP. This result suggests that SSP's potential is much higher if the CPU overhead of communication were reduced in our current implementation.

Config.	Bytes sent	Bytes received
wpc=4, slack=0	33.0 M	29.7 M
wpc=2, slack=1	61.9 M	51.0 M
wpc=1, slack=3	119.4 M	81.5 M

Table 2: Bytes sent/rec'd per client per iteration of TM.

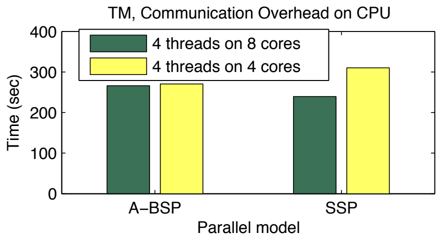


Figure 8: CPU overhead of communication.

5.5 Prefetching and throughput

Figure 9 shows the importance of prefetching, highlighting the value of LazyTable's iteration-aware prefetching scheme. The time per iteration, partitioned into computation time and wait time, is shown for each of the three applications using SSP. In each case, the prefetching significantly reduces the wait time. There are two reasons that the speed up for MF is higher than for TM and PR. First, the set of rows accessed by different application threads overlap less in MF than in TM; so, when there is no prefetching, the rows used by one thread on a machine are less likely to be already fetched by another thread and put into the shared process cache. Second, each iteration is longer in MF, which means the same slack value covers more work; so, with prefetching, the threads are less likely to have data misses.

Note that, as expected, prefetching reduces wait time but not computation time. An interesting lesson we learned when prefetching was added to LazyTable is that it tends to reduce the best-choice staleness bound. Because increased staleness reduces wait time, prefetching's tendency to reduce wait time reduces the opportunity to increase iteration speed. So, the iteration effectiveness component of convergence speed plays a larger role in determining the best staleness bound. Before adding prefetching, we observed that higher staleness bounds were better than those reported here.

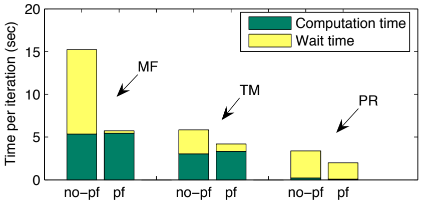


Figure 9: Time per iteration with/without prefetching for all three applications with {wpc=1, slack=1}.

6 Related work

In a HotOS workshop paper [12], we proposed SSP, briefed an early LazyTable prototype that implemented it, and did a couple of experiments to show that it helps mitigate delayed thread stragglers. We recently published follow-on work in an ML conference [18] that provides evidence of convergence for several ML algorithms under SSP, including proofs that provide theoretical justification for SSP's bounded staleness model. Some experiments comparing SSP to BSP show performance improvement, but the focus is on the ML algorithm behavior. This paper makes several important contributions beyond our previous papers: we describe a more general view of bounded staleness covering A-BSP as well as SSP, we describe in detail a system that supports both, we explain design details that are important to realizing their performance potential, and we thoroughly evaluate both and show the strengths and weaknesses for each. Importantly, whereas SSP almost always outperforms BSP (as commonly used in ML work) significantly, this paper makes clear that A-BSP can be as effective when straggler issues are minor.

The High Performance Computing community – which frequently runs applications using the BSP model – has made much progress in eliminating stragglers caused by hardware or operating system effects [13, 27]. While these solutions are very effective at reducing "operating system jitter", they are not intended to solve the more general straggler problem. For instance, they are not applicable to programs written in garbage collected languages, nor do they handle algorithms that inherently cause stragglers during some iterations.

In large-scale networked systems, where variable node performance, unpredictable communication latencies and failures are the norm, researchers have explored relaxing traditional barrier synchronization. For example, Albrecht et al. [2] describe partial barriers, which allow a fraction of nodes to pass through a barrier by adapting the rate of entry and release from the barrier. This approach does not bound how far behind some nodes may get, which is important for ML algorithm convergence.

Another class of solutions attempts to reduce the need for synchronization by restricting the structure of the communication patterns. For example, GraphLab [25, 26] programs structure computation as a graph, where data can exist on nodes and edges. All communication occurs along the edges of this graph. If two nodes on the graph are sufficiently far apart they may be updated without synchronization. This model can significantly reduce synchronization in some cases. However, it requires the application programmer to specify the communication pattern explicitly.

Considerable work has been done in describing and enforcing relaxed consistency in distributed replicated services. For example, the TACT model [31] describes consistency along three dimensions: numerical error, order error and staleness. Other work [30] classifies existing systems according to a number of consistency properties, specifically naming the concept of bounded staleness. Although the context differs, the consistency models have some similarities.

In the database literature, bounded staleness has been applied to improve update and query performance. LazyBase [11] allows staleness bounds to be configured on a per-query basis, and uses this relaxed staleness to improve both query and update performance. FAS [29] keeps data replicated in a number of databases, each providing a different freshness/performance tradeoff. Data stream warehouses [16] collect data about timestamped events, and provide different consistency depending on the freshness of the data. The concept of staleness (or freshness/timeliness) has also been applied in other fields such as sensor networks [20], dynamic web content generation [22], web caching [9], and information systems [7].

Of course, one can ignore consistency and synchronization altogether, relying on a best-effort model for updating shared data. Yahoo! LDA [1] as well as most solutions based around NoSQL databases rely on this model. While this approach can work well in some cases, having no staleness bounds makes confidence in ML algorithm convergence difficult.

7 Conclusion

Bounded staleness reduces communication and synchronization overheads, allowing parallel ML algorithms to converge more quickly. LazyTable supports parallel ML execution using any of BSP, A-BSP, or SSP. Experiments with three ML applications executed on 500 cores show that both A-BSP and SSP are effective in the absence of stragglers. SSP mitigates stragglers more effectively, making it the best option in environments with more variability, such as clusters with multiple uses and/or many software layers, or for algorithms with more variability in the work done per thread within an iteration.

Acknowledgements. We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-IO, Google, Hewlett-Packard, Hitachi, Huawei, Intel, Microsoft, NEC Labs, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, Western Digital). This research is supported in part by the Intel Science and Technology Center for Cloud Computing (ISTC-CC), National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE [15]), and DARPA Grant FA87501220324.

References

[1] Ahmed, A., Aly, M., Gonzalez, J., Narayanamurthy, S., and Smola, A. J. Scalable inference in latent variable models. In *WSDM* (2012).

[2] Albrecht, J., Tuttle, C., Snoeren, A. C., and Vahdat, A. Loose synchronization for large-scale networked systems. In *USENIX Annual Tech* (2006).

[3] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the outliers in map-reduce clusters using Mantri. In *OSDI* (2010).

[4] Apache Mahout, http://mahout.apache.org.

[5] Beckman, P., Iskra, K., Yoshii, K., and Coghlan, S. The influence of operating systems on the performance of collective operations at extreme scale. *CLUSTER* (2006).

[6] Blei, D. M., Ng, A. Y., and Jordan, M. I. Latent dirichlet allocation. *JMLR* (2003).

[7] Bouzeghoub, M. A framework for analysis of data freshness. In *IQIS* (2004).

[8] Bradley, J. K., Kyrola, A., Bickson, D., and Guestrin, C. Parallel coordinate descent for L1-regularized loss minimization. In *ICML* (2011).

[9] Bright, L., and Raschid, L. Using latency-recency profiles for data delivery on the web. In *VLDB* (2002).

[10] Brin, S., and Page, L. The anatomy of a large-scale hypertextual web search engine. *Computer Networks* (1998).

[11] Cipar, J., Ganger, G., Keeton, K., Morrey III, C. B., Soules, C. A., and Veitch, A. LazyBase: Trading freshness for performance in a scalable database. In *Eurosys* (2012).

[12] Cipar, J., Ho, Q., Kim, J. K., Lee, S., Ganger, G. R., Gibson, G., Keeton, K., and Xing, E. Solving the straggler problem with bounded staleness. In *HotOS* (2013).

[13] Ferreira, K. B., Bridges, P. G., Brightwell, R., and Pedretti, K. T. The impact of system design parameters on application noise sensitivity. In *CLUSTER* (2010).

[14] Gemulla, R., Nijkamp, E., Haas, P. J., and Sismanis, Y. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD* (2011).

[15] Gibson, G., Grider, G., Jacobson, A., and Lloyd, W. PRObE: A thousand-node experimental cluster for computer systems research. *USENIX ;login:* (2013).

[16] Golab, L., and Johnson, T. Consistency in a stream warehouse. In *CIDR* (2011).

[17] Griffiths, T. L., and Steyvers, M. Finding scientific topics. *Proc. National Academy of Sciences USA* (2004).

[18] Ho, Q., Cipar, J., Cui, H., Kim, J. K., Lee, S., Gibbons, P. B., Gibson, G. A., Ganger, G. R., and Xing, E. P. More effective distributed ML via a stale synchronous parallel parameter server. In *NIPS* (2013).

[19] Hoffman, M., Bach, F. R., and Blei, D. M. Online learning for latent dirichlet allocation. In *NIPS* (2010).

[20] Huang, C.-T. Loft: Low-overhead freshness transmission in sensor networks. In *SUTC* (2008).

[21] Krevat, E., Tucek, J., and Ganger, G. R. Disks are like snowflakes: No two are alike. In *HotOS* (2011).

[22] Labrinidis, A., and Roussopoulos, N. Balancing performance and data freshness in web database servers. In *VLDB* (2003).

[23] Langford, J., Smola, A. J., and Zinkevich, M. Slow learners are fast. In *NIPS* (2009).

[24] Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* (2009).

[25] 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).

[26] Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., and Hellerstein, J. M. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *PVLDB* (2012).

[27] Petrini, F., Kerbyson, D. J., and Pakin, S. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Supercomputing* (2003).

[28] Power, R., and Li, J. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI* (2010).

[29] Röhm, U., Böhm, K., Schek, H.-J., and Schuldt, H. FAS: A freshness-sensitive coordination middleware for a cluster of OLAP components. In *VLDB* (2002).

[30] Terry, D. Replicated data consistency explained through baseball. Tech. rep., Microsoft Research, 2011.

[31] Yu, H., and Vahdat, A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS* (2002).

[32] ZeroMQ: The intelligent transport layer. http://www.zeromq.org/.

Making State Explicit for Imperative Big Data Processing

Raul Castro Fernandez*, Matteo Migliavacca†, Evangelia Kalyvianaki♯, Peter Pietzuch*
*Imperial College London, †University of Kent, ♯City University London

Abstract

Data scientists often implement machine learning algorithms in imperative languages such as Java, Matlab and R. Yet such implementations fail to achieve the performance and scalability of specialised data-parallel processing frameworks. Our goal is to execute imperative Java programs in a data-parallel fashion with high throughput and low latency. This raises two challenges: how to support the arbitrary mutable state of Java programs without compromising scalability, and how to recover that state after failure with low overhead.

Our idea is to infer the dataflow and the types of state accesses from a Java program and use this information to generate a *stateful dataflow graph (SDG)*. By explicitly separating *data* from mutable *state*, SDGs have specific features to enable this translation: to ensure scalability, distributed state can be *partitioned* across nodes if computation can occur entirely in parallel; if this is not possible, *partial* state gives nodes local instances for independent computation, which are reconciled according to application semantics. For fault tolerance, large in-memory state is checkpointed asynchronously without global coordination. We show that the performance of SDGs for several imperative online applications matches that of existing data-parallel processing frameworks.

1 Introduction

Data scientists want to use ever more sophisticated implementations of machine learning algorithms, such as collaborative filtering [32], k-means clustering and logistic regression [21], and execute them over large datasets while providing “fresh”, low latency results. With the dominance of imperative programming, such algorithms are often implemented in languages such as Java, Matlab or R. Such implementations though make it challenging to achieve high performance.

On the other hand, data-parallel processing frameworks, such as MapReduce [8], Spark [38] and Naiad [26], can scale computation to a large number of nodes. Such frameworks, however, require developers to adopt particular functional [37], declarative [13] or dataflow [15] programming models. While early frameworks such as MapReduce [8] followed a restricted functional model, resulting in wide-spread adoption, recent more expressive frameworks such as Spark [38] and Naiad [26] require developers to learn more complex programming models, e.g. based on a richer set of higher-order functions.

Our goal is therefore to translate imperative Java implementations of machine learning algorithms to a representation that can be executed in a data-parallel fashion. The execution should scale to a large number of nodes, achieving high throughput and low processing latency. This is challenging because Java programs support arbitrary mutable state. For example, an implementation of collaborative filtering [32] uses a mutable matrix to represent a model that is refined iteratively: as new data arrives, the matrix is updated at a fine granularity and accessed to provide up-to-date predictions.

Having stateful computation raises two issues: first, the state may grow large, e.g. on the order of hundreds of GBs for a collaborative filtering model with tens of thousands of users. Therefore the state and its associated computation must be distributed across nodes; second, large state must be restored efficiently after node failure. The failure recovery mechanism should have a low impact on performance.

Current data-parallel frameworks do not handle large state effectively. In stateless frameworks [8, 37, 38], computation is defined through side-effect-free functional tasks. Any modification to state, such as updating a single element in a matrix, must be implemented as the creation of new immutable data, which is inefficient. While recent frameworks [26, 10] have recognised the need for per-task mutable state, they lack abstractions for distributed state and exhibit high overhead under fault-tolerant operation with large state (see §6.1).

Imperative programming model. We describe how, with the help of a few annotations by developers, Java programs can be executed automatically in a distributed data-parallel fashion. Our idea is to infer the dataflow and the types of state accesses from a Java program and use this information to translate the program to an executable distributed dataflow representation. Using program analysis, our approach extracts the processing tasks and state fields from the program and infers the variable-level dataflow.

Stateful dataflow graphs. This translation relies on the features of a new fault-tolerant data-parallel processing model called *stateful dataflow graphs (SDGs)*. An SDG explicitly distinguishes between *data* and *state*: it is a cyclic graph of pipelined data-parallel tasks, which execute on different nodes and access local in-memory state.

SDGs include abstractions for maintaining large state efficiently in a distributed fashion: if tasks can process state entirely in parallel, the state is *partitioned* across nodes; if this is not possible, tasks are given local instances of *partial* state for independent computation. Computation can include synchronisation points to access all partial state instances, and instances can be reconciled according to application semantics.

Data flows between tasks in an SDG, and cycles specify iterative computation. All tasks are pipelined—this leads to low latency, less intermediate data during failure recovery and simplified scheduling by not having to compute data dependencies. Tasks are replicated at runtime to overcome processing bottlenecks and stragglers.

Failure recovery. When recovering from failures, nodes must restore potentially gigabytes of in-memory state. We describe an asynchronous checkpointing mechanism with log-based recovery that uses data structures for dirty state to minimise the interruption to tasks while taking local checkpoints. Checkpoints are persisted to multiple disks in parallel, from which they can be restored to multiple nodes, thus reducing recovery time.

With a prototype system of SDGs, we execute Java implementations of collaborative filtering, logistic regression and a key/value store on a private cluster and Amazon EC2. We show that SDGs execute with high throughput (comparable to batch processing systems) and low latency (comparable to streaming systems). Even with large state, their failure recovery mechanism has a low performance impact, recovering in seconds.

The paper contributions and its structure are as follows: based on a sample Java program (§2.1) and the features of existing dataflow models (§2.2), we motivate the need for stateful dataflow graphs and describe their properties (§3); §4 explains the translation from Java to SDGs; §5 describes failure recovery; and §6 presents evaluation results, followed by related work (§7).

Algorithm 1: Online collaborative filtering

```
1  @Partitioned Matrix userItem = new Matrix();
2  @Partial Matrix coOcc = new Matrix();
3
4  void addRating(int user, int item, int rating) {
5    userItem.setElement(user, item, rating);
6    Vector userRow = userItem.getRow(user);
7    for (int i = 0; i < userRow.size(); i++)
8      if (userRow.get(i) > 0) {
9        int count = coOcc.getElement(item, i);
10       coOcc.setElement(item, i, count + 1);
11       coOcc.setElement(i, item, count + 1);
12     }
13 }
14 Vector getRec(int user) {
15   Vector userRow = userItem.getRow(user);
16   @Partial Vector userRec = @Global coOcc.multiply(
          userRow);
17   Vector rec = merge(@Global userRec);
18   return rec;
19 }
20 Vector merge(@Collection Vector[] allUserRec) {
21   Vector rec = new Vector(allUserRec[0].size());
22   for (Vector cur : allUserRec)
23     for (int i = 0; i < allUserRec.length; i++)
24       rec.set(i, cur.get(i) + rec.get(i));
25   return rec;
26 }
```

2 State in Data-Parallel Processing

We describe an imperative implementation of a machine learning algorithm and investigate how it can execute in a data-parallel fashion on a set of nodes, paying attention to its use of mutable state (§2.1). Based on this analysis, we discuss the features of existing data-parallel processing models for supporting such an execution (§2.2).

2.1 Application example

Alg. 1 shows a Java implementation of an online machine learning algorithm, *collaborative filtering* (CF) [32].[1] It outputs up-to-date recommendations of items to users (function getRec) based on previous item ratings (function addRating).

The algorithm maintains state in two data structures: the matrix userItem stores the ratings of items made by users (line 1); the co-occurrence matrix coOcc records correlations between items that were rated together by multiple users (line 2).

For many users and items, useritem and coOcc become large and must be distributed: userItem can be *partitioned* across nodes based on the user identifier as an access key; since the access to coOcc is random, it cannot be partitioned but only *replicated* on multiple nodes in order to parallelise updates. In this case, results from a single instance of coOcc are *partial*, and must be merged with other partial results to obtain a complete result, as described below.

The function addRating first adds a new rating to userItem (line 5). It then incrementally updates coOcc by increasing the co-occurrence counts for the newly-rated

[1] The annotations (starting with '@') will be explained in §4 and should be ignored for now.

Computational model	*Systems*	*Programming model*	*State handling*			*Dataflow*			*Failure recovery*
			Representation	*Large state size*	*Fine-grained updates*	*Execution*	*Low latency*	*Iteration*	
Stateless dataflow	MapReduce [8]	map/reduce	as data	n/a	✗	scheduled	✗	✗	recompute
	DryadLINQ [37]	functional	as data	n/a	✗	scheduled	✗	✓	recompute
	Spark [38]	functional	as data	n/a	✗	hybrid	✗	✓	recompute
	CIEL [25]	imperative	as data	n/a	✗	scheduled	✗	✓	recompute
	HaLoop [5]	map/reduce	cache	✓	✗	scheduled	✗	✓	recompute
Incremental dataflow	Incoop [4]	map/reduce	cache	✓	✗	scheduled	✗	✗	recompute
	Nectar [11]	functional	cache	✓	✗	scheduled	✗	✗	recompute
	CBP [19]	dataflow	loopback	✓	✓	scheduled	✗	✗	recompute
Batched dataflow	Comet [12]	functional	as data	n/a	✗	scheduled	✓	✗	recompute
	D-Streams [39]	functional	as data	n/a	✗	hybrid	✓	✓	recompute
	Naiad [26]	dataflow	explicit	✗	✓	hybrid	✓	✓	sync. global checkpoints
Continuous dataflow	Storm, S4	dataflow	as data	n/a	✗	pipelined	✓	✗	recompute
	SEEP [10]	dataflow	explicit	✗	✓	pipelined	✓	✗	sync. local checkpoints
Parallel in-memory	Piccolo [30]	imperative	explicit	✓	✓	n/a	✓	✓	async. global checkpoints
Stateful dataflow	SDG	imperative	explicit	✓	✓	pipelined	✓	✓	async. local checkpoints

Table 1: Design space of data-parallel processing frameworks

`item` and existing items with non-zero ratings (line 7–12). This requires `userItem` and `coOcc` to be mutable, with efficient fine-grained access. Since `userItem` is partitioned based on the key `user`, and `coOcc` is replicated, `addRating` only accesses a single instance of each.

The function `getRec` takes the rating vector of a user, `userRow` (line 15), and multiplies it by the co-occurrence matrix to obtain a recommendation vector `userRec` (line 16). Since `coOcc` is replicated, this must be performed on *all* instances of `coOcc`, leading to multiple partial recommendation vectors. These partial vectors must be *merged* to obtain the final recommendation vector `rec` for the user (line 17). The function `merge` simply computes the sum of all partial recommendation vectors (lines 21–24).

Note that `addRating` and `getRec` have different performance goals when handling state: `addRating` must achieve *high throughput* when updating `coOcc` with new ratings; `getRec` must serve requests with *low latency*, e.g. when recommendations are included in dynamically generated web pages.

2.2 Design space

The above example highlights a number of required features of a dataflow model to enable the translation of imperative online machine learning algorithms to executable dataflows: (i) the model should support *large state sizes* (on the order of GBs), which should be represented explicitly and handled with acceptable performance; in particular, (ii) the state should permit efficient *fine-grained updates*. In addition, due to the need for up-to-date results, (iii) the model should process data with *low latency*, independently of the amount of input data; (iv) algorithms such as logistic regression and k-means clustering also require *iteration*; and (v) even with large state, the model should support fast *failure recovery*.

In Table 1, we classify existing data-parallel processing models according to the above features.

State handling. *Stateless dataflows*, first made popular by MapReduce [8], define a functional dataflow graph in which vertices are stateless data-parallel tasks. They do not distinguish between state and data: e.g. in a word-count job in MapReduce, the partial word counts, which are the state, are output by map tasks as part of the dataflow [8]. Dataflows in Spark, represented as RDDs, are immutable, which simplifies failure recovery but requires a new RDD for each state update [38]. This is inefficient for online algorithms such as CF in which only part of a matrix is updated each time.

Stateless models also cannot treat data differently from state. They cannot use custom index data structures for state access, or cache only state in memory: e.g. Shark [36] needs explicit hints which dataflows to cache.

Incremental dataflow avoids rerunning entire jobs after updates to the input data. Such models are fundamentally stateful because they maintain results from earlier computation. Incoop [4] and Nectar [11] treat state as a cache of past results. Since they cannot infer which data will be reused, they cache all. CBP transforms batch jobs automatically for incremental computation [19].

Our goals are complementary: SDGs do not infer incremental computation but support stateful computation efficiently, which can realise incremental algorithms.

Existing models that represent state explicitly, such as SEEP [10] and Naiad [26], permit tasks to have access to in-memory data structures but face challenges related to state sizes: they assume that state is small compared to the data. When large state requires distributed processing through partitioning or replication, they do not provide abstractions to support this.

In contrast, Piccolo [30] supports scalable distributed state with a key/value abstraction. However, it does not offer a dataflow model, which means that it cannot execute an inferred dataflow from a Java program but requires computation to be specified as multiple kernels.

Latency and iteration. Tasks in a dataflow graph can be *scheduled* for execution or materialised in a *pipeline*, each with different performance implications. Some frameworks follow a *hybrid* approach in which tasks on the same node are pipelined but not between nodes.

Since tasks in *stateless dataflows* are scheduled to process coarse-grained batches of data, such systems can exploit the full parallelism of a cluster but they cannot achieve low processing latency. For lower latency, *batched dataflows* divide data into small batches for processing and use efficient, yet complex, task schedulers to resolve data dependencies. They have a fundamental trade-off between the lower latency of smaller batches and the higher throughput of larger ones—typically they burden developers with making this trade-off [39].

Continuous dataflow adopts a streaming model with a pipeline of tasks. It does not materialise intermediate data between nodes and thus has lower latency without a scheduling overhead: as we show in §6, batched dataflows cannot achieve the same low latencies. Due to our focus on online processing with low latency, SDGs are fully pipelined (see §3.1).

To improve the performance of *iterative computation* in dataflows, early frameworks such as HaLoop [5] cache the results of one iteration as input to the next. Recent frameworks [15, 38, 25, 9] generalise this concept by permitting iteration over arbitrary parts of the dataflow graph, executing tasks repeatedly as part of loops. Similarly SDGs support iteration explicitly by permitting cycles in the dataflow graph.

Failure recovery. To recover from failure, frameworks either recompute state based on previous data or checkpoint state to restore it. For *recomputation*, Spark represents dataflows as RDDs [38], which can be recomputed deterministically based on their lineage. *Continuous dataflow* frameworks use techniques such as *upstream backup* [14] to reprocess buffered data after failure. Without checkpointing, recomputation can lead to long recovery times.

Checkpointing periodically saves state to disk or the memory of other nodes. With large state, this becomes resource-intensive. SEEP recovers state from memory, thus doubling the memory requirement of a cluster [10].

A challenge is how to take consistent checkpoints while processing data. *Synchronous global* checkpointing stops processing on all nodes to obtain consistent snapshots, thus reducing performance. For example, Naiad's "stop-the-world" approach exhibits low throughput with large state sizes [26]. *Asynchronous global* checkpointing, as used by Piccolo [30], permits nodes to take consistent checkpoints at different times.

Both techniques include all global state in a checkpoint and thus require all nodes to restore state after failure. Instead, SDGs use an asynchronous checkpointing mechanism with log-based recovery. As described in §5, it does not require global coordination between nodes during recovery, and it uses dirty state to minimise the disruption to processing during local checkpointing.

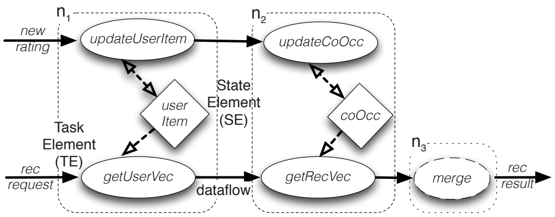


Figure 1: Stateful dataflow graph for CF algorithm

3 Stateful Dataflow Graphs

The goal of *stateful dataflow graphs* (SDGs) is to make it easy to translate imperative programs with mutable state to a dataflow representation that performs parallel, iterative computation with low latency. Next we describe their model (§3.1), how they support distributed state (§3.2) and how they are executed (§3.3).

3.1 Model

We explain the main features of SDGs using the CF algorithm from §2.1 as an example. As shown in Fig. 1, an SDG has two types of vertices: *task elements*, $t \in T$, transform input to output dataflows; and *state elements*, $s \in S$, represent the state in the SDG.

Access edges, $a = (t, s) \in A$, connect task elements to the state elements that they read or update. To facilitate the allocation of task and state elements to nodes, each task element can only access a single state element, i.e. A is a partial function: $(t_i, s_j) \in A, (t_i, s_k) \in A \Rightarrow s_j = s_k$. *Dataflows* are edges between task elements, $d = (t_i, t_j) \in D$, and contain data items.

Task elements (TEs) are not scheduled for execution but the entire SDG is materialised, i.e. each TE is assigned to one or more physical nodes. Since TEs are pipelined, it is unnecessary to generate the complete output dataflow of a TE before it is processed by the next TE. Data items are therefore processed with low latency, even across a sequence of TEs, without scheduling overhead, and fewer data items are handled during failure recovery (see §5).

The SDG in Fig. 1 has five TEs assigned to three nodes: the `updateUserItem`, `updateCoOcc` TEs realise the `addRating` function from Alg. 1; and the `getUserVec`, `getRecVec` and `merge` TEs implement the `getRec` function. We explain the translation process in §4.2.

State elements (SEs) encapsulate the state of the computation. They are implemented using efficient data structures, such as hash tables or indexed sparse matrices. In the next section, we describe the abstractions for distributed SEs, which span multiple nodes.

Fig. 1 shows the two SEs of the CF algorithm: the `userItem` and the `coOcc` matrices. The access edges specify that userItem is updated by the updateUserItem TE and read by the getUserVec TE; coOcc is updated by updateCoOcc and read by getRecVec.

Parallelism. For data-parallel processing, a TE t_i can be instantiated multiple times to handle parts of a dataflow, resulting in multiple TE instances, $\hat{t}_{i,j} : j \leq n_i$. As we explain in §3.3, the number of instances n_i for each TE is chosen at runtime and adjusted based on workload demands and the occurrence of stragglers.

An appropriate dispatching strategy sends items in dataflows to TE instances: items can be (i) partitioned using hash- or range-partitioning on a key; or (ii) dispatched to an *arbitrary* instance, e.g. in a round-robin fashion for load-balancing.

Iteration. In iterative algorithms, SEs are accessed multiple times by TEs. There are two cases to be distinguished: (i) if the repeated access is from a single TE, the iteration is entirely local and can be supported efficiently by a single node; and (ii) if the iteration involves multiple pipelined TEs, a *cycle* in the dataflow of the SDG can propagate updates between TEs.

With cycles in the dataflow, SDGs do not provide coordination during iteration by default. This is sufficient for many iterative machine learning and data mining algorithms because they can converge from different intermediate states [31], even without explicit coordination. A strong consistency model for SDGs could be realised with per-loop timestamps, as used by Naiad [26].

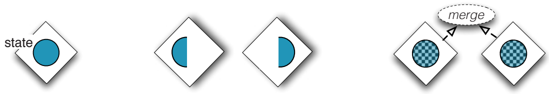
3.2 Distributed state

The SDG model provides abstractions for distributed state. An SE s_i may be distributed across nodes, leading to multiple SE instances $\hat{s}_{i,j}$, because (i) it is too large to fit into the memory of a single node; or (ii) it is accessed by a TE that has multiple instances to process the dataflow in parallel. This requires also multiple SE instances so that the TE instances access state locally.

Fig. 1 illustrates these two cases: (i) the userItem SE may grow larger than the main memory of a single node; and (ii) the data-parallel execution of the CPU-intensive updateCoOcc TE leads to multiple instances, each requiring local access to the coOcc SE.

An SE can be distributed in different ways, which are depicted in Fig. 2: a *partitioned* SE splits its internal data structure into disjoint partitions; if this is not possible, a *partial* SE duplicates its data structure, creating multiple copies that are updated independently. As we describe in §4, developers selected the required type of distributed state using source-level annotations according to the semantics of their algorithm.

Partitioned state. For algorithms for which state can be partitioned, SEs can be split and SE instances placed on separate nodes (see Fig. 2b). Access to the SE instances occurs in parallel.



(a) SE (b) Partitioned SE (c) Partial SE

Figure 2: Types of distributed state in SDGs

Developers can use predefined data structures for SEs (e.g. Vector, HashMap, Matrix and DenseMatrix) or define their own by implementing dynamic partitioning and dirty state support (see §5). Different data structures support different partitioning strategies: e.g. a map can be hash- or range-partitioned; a matrix can be partitioned by row or column. To obtain a unique partitioning, TEs cannot access partitioned SEs using conflicting strategies, such as accessing a matrix by row and by column.

In addition, the dataflow partitioning strategy must be compatible with the data access pattern by the TEs, as specified in the program (see §4.2). For example, multiple TE instances with an access edge to a partitioned SE must use the same partitioning key on the dataflow so that they access SE instances locally: in the CF algorithm, the userItem SE and the new rating and rec request dataflows must all be partitioned by row, i.e. the users for which ratings are maintained.

Partial state. In some cases, the data structure of an SE cannot be partitioned because the access pattern of TEs is arbitrary. For example, in the CF algorithm, the coOcc matrix has an access pattern, in which the updateCoOcc TE may update any row or column. In this case, an SE is distributed by creating multiple *partial* SE instances, each containing the whole data structure (see Fig. 2c). Partial SE instances can be updated independently by different TE instances.

When a TE accesses a partial SE, there are two possible types of accesses based on the semantics of the algorithm: a TE instance may access (i) the *local* SE instance on the same node; or (ii) the *global* state by accessing all of the partial SE instances, which introduces a synchronisation point. As we describe in §4.2, the type of access to partial SEs is determined by annotations.

When accessing all partial SE instances, it is possible to execute computation that *merges* their values, thus reconciling the differences between them. This is done by a merge TE that computes a single global value from partial SE instances. Merge computation is application-specific and must be defined by the developer. In the CF algorithm, the merge function takes all partial userRec vectors and computes a single recommendation vector.

3.3 Execution

To execute an SDG, the runtime system allocates TE and SE instances to nodes, creating instances on-demand.

Allocation to nodes. Since we want to avoid remote state access, the general strategy is to colocate TEs and

SEs that are connected by access edges on the same node. The runtime system uses four steps for mapping TEs and SEs to nodes: if there is a cycle in the SDG, all SEs accessed in the cycle are colocated if possible to reduce communication in iterative algorithms (step 1); the remaining SEs are allocated on separate nodes to increase available memory (step 2); TEs are colocated with the SEs that they access (step 3); and finally, any unallocated TEs are assigned to separate nodes (step 4).

Fig. 1 illustrates the above steps for allocating the SDG to nodes n_1 to n_3: since there are no cycles (step 1), the `userItem` SE is assigned to node n_1, and the `coOcc` SE is assigned to n_2 (step 2); the `updateUserItem` and `getUserVec` TEs are assigned to n_1, and the `updateCoOcc` and `getRecVec` TEs are assigned to n_2 (step 3); finally, the `merge` TE is allocated to a new node n_3 (step 4).

Runtime parallelism and stragglers. Processing bottlenecks in the deployed SDG, e.g. caused by the computational cost of TEs, cannot be predicted statically, and TEs instances may become stragglers [40]. Previous work [26] tries to reduce stragglers *proactively* for low latency processing, which is hard due to the many non-deterministic causes of stragglers.

Instead, similar to speculative execution in MapReduce [40], SDGs adopt a *reactive* approach. Using a dynamic dataflow graph approach [10], the runtime system changes the number of TE instances in response to stragglers. Each TE is monitored to determine if it constitutes a processing bottleneck that limits throughput. If so, a new TE instance is created, which may result in new partitioned or partial SE instances.

3.4 Discussion

With an explicit representation of state, a single SDG can express multiple workflows over that state. In the case of the CF algorithm from Alg. 1, the SDG processes new ratings by updating the SEs for the user/item and co-occurrence matrices, and also serves recommendation requests using the same SEs with low latency.

Without SDGs, these two workflows would require separate offline and online systems [23, 32]: a batch processing framework would incorporate new ratings periodically, and online recommendation requests would be served by a dedicated system from memory. Since it is inefficient to rerun the batch job after each new rating, the recommendations would be computed on stale data.

A drawback of the materialised representation of SDGs is the start-up cost. For short jobs, the deployment cost may dominate the running time. Our prototype implementation deploys an SDG with 50 TE and SE instances on 50 nodes within 7 s, and we assume that jobs are sufficiently long-running to amortise this delay.

4 Programming SDGs

We describe how to translate stateful Java programs statically to SDGs for parallel execution. We do not attempt to be completely transparent for developers or to address the general problem of automatic code parallelisation. Instead, we exploit data and pipeline parallelism by relying on source code annotations. We require developers to provide a single Java class with annotations that indicate how state is distributed and accessed.

4.1 Annotations

When defining a field in a Java class, a developer can indicate if its content can be *partitioned* or is *partial* by annotating the field declaration with `@Partitioned` or `@Partial`, respectively.

`@Partitioned`. This annotation specifies that a field can be split into disjoint partitions (see §3.2). A reference to a `@Partitioned` field always refers to a single partition. This requires that access to the field uses an access key to infer the partition. In the CF algorithm in Alg. 1, rows of the `userItem` matrix are updated with information about a single user only, and thus `userItem` can be declared as a partitioned field.

`@Partial`. Fields are annotated with `@Partial` if distributed instances of the field should be accessed independently (see §3.2). Partial fields enable developers to define distributed state when it cannot be partitioned. In CF, matrix `coOcc` is annotated with `@Partial`, which means that multiple instances of the matrix may be created, and each of them is updated independently for users in a partition (lines 10–11).

`@Global`. By default, a reference to a `@Partial` field refers to only one of its instances. While most of the time, computation should apply to one instance to make independent progress, it may also be necessary to support operations on *all* instances. A field reference annotated with `@Global` forces a Java expression to apply to all instances, denoting "global" access to a partial field, which introduces a synchronisation barrier in the SDG (see §4.2).

Java expressions deriving from `@Global` access become logically *multi-valued* because they include results from all instances of a partial field. As a result, any local variable that is assigned the result of a global field access becomes partial and must be annotated as such.

In CF, the access to the `coOcc` field carries the `@Global` annotation to compute *all* partial recommendations: each instance of `coOcc` is multiplied with the user rating vector `userRow`, and the results are stored in the partial local variable `userRec` (line 16).

`@Collection`. Global access to a partial field applies to all instances, but it hides the individual instances from the developer. At some point in the program, however, it may be necessary to reconcile all instances. The

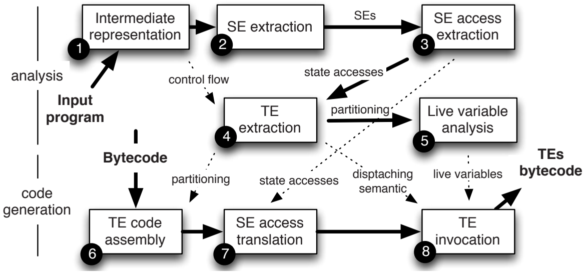


Figure 3: Translation of an annotated Java program to an SDG

`@Collection` annotation therefore exposes all instances of a partial field or variable as a Java array after `@Global` access. This enables the program to iterate over all values and, for example, merge them into a single value.

In CF, the partial recommendations are combined by accessing them using the `@Global` annotation and then invoking the `merge` method (line 17). The parameter of `merge` is annotated with `@Collection`, which specifies that the method can access all instances of the partial `userRec` variable to compute the final recommendation result.

Limitations. Java programs need to obey certain restrictions to be translated to SDGs due to their dataflow nature and fault tolerance properties:

Explicit state classes. All state in the program must be implemented using the set of SE classes (see §3.2). This gives the runtime system the ability to partition objects of these classes into multiple instances (for partitioned state) or distribute them (for partial state), and recover them after failure (see §5).

Location independence. Each object accessed in the program must support transparent serialisation/deserialisation: as SDGs are distributed, objects are propagated between nodes. The program also cannot make assumptions about its execution environment, e.g. by relying on local network sockets or files.

Side-effect-free parallelism. To support the parallel evaluation of multi-valued expressions under `@Global` state access, such expressions must not affect single-valued expressions. For example, the statement, `@Global coOcc.multiply(userRow)`, in line 16 in Alg. 1 cannot update `userRow`, which is single-valued.

Deterministic execution. The program must be deterministic, i.e. it should not depend on system time or random input. This enables the runtime system to re-execute computation when recovering after failure (see §5).

4.2 Translating programs to SDGs

Annotated Java programs are translated to SDGs by the `java2sdg` tool. Fig. 3 shows the steps performed by `java2sdg`: it first statically analyses the Java class to identify SEs, TEs and their access edges (steps 1–5); it then transforms the Java bytecode of the class to generate TE code, ready for deployment (steps 6–8).

1. SE generation. The class is compiled to *Jimple* code, a typed intermediate representation for static analysis used by the *Soot* framework [33] (step 1). The Jimple code is analysed to identify SEs with partitioned or partial fields and partial local variables (step 2). Based on the annotations in the code, access to SEs is classified as local, partitioned or global (step 3).

2. TE and dataflow generation. Next TEs are created so that each TE only accesses a single SE, i.e. a new TE is created from a block of code when access to a different SE or a different instance of the current SE is detected (step 4). The dispatching semantics of the dataflows between created TEs (i.e. partitioned, all-to-one, one-to-all or one-to-any) is chosen based on the type of state access. More specifically, a new TE is created:

1. for each entry point of the class;
2. when a TE uses *partitioned access* to a new SE (or to a previously-accessed SE with a new access key). The access key is extracted using reaching expression analysis, and the dataflow edge between the two TEs is annotated with the access key;
3. when a TE uses *global access* to a new partial SE. In this case, the dataflow edge between the two TEs is annotated with *one-to-all* dispatching semantics;
4. when a TE uses *local access* to a new partial SE, the dataflow edge is annotated with *one-to-any* dispatching semantics. In case of local (or partitioned) access after global access, all TE instances must be synchronised using a distributed barrier before control is transferred to the new TE, and the dataflow edge has *all-to-one* dispatching semantics; and
5. for `@Collection` expressions. A synchronisation barrier collects values from multiple TEs instances, and its dataflow edge has *all-to-one* semantics.

After generating the TEs, `java2sdg` identifies the variables that must propagate across TEs boundaries (step 5). For each dataflow, live variable analysis identifies the set of variables that are associated with that dataflow edge.

3. Bytecode generation. Next `java2sdg` synthesises the bytecode for each TE that will be executed by the runtime system. It compiles the code assigned with each TE in step 4 to bytecode and injects it into a TE template (step 6) using *Javassist*. State accesses to fields and partial variables are translated to invocations of the runtime system, which manages the SE instances (step 7).

Finally data dispatching across TEs is added (step 8): `java2sdg` injects code, (i) at the exit point of TEs, to serialise live variables and send them to the correct successor TE instance; and, (ii) at the entry point of a TE, to add barriers for all-to-one dispatching and to gather partial results for merge TEs.

5 Failure Recovery

To recover from failures, it is necessary to replace failed nodes and re-instantiate their TEs and SEs. TEs are stateless and thus are restored trivially, but the state of SEs must be recovered. We face the challenge of designing a recovery mechanism that: (i) can scale to save and recover the state of a large number of nodes with low overhead, even with frequent failures; (ii) has low impact on the processing latency; and (iii) achieves fast recovery time when recovering large SEs.

We achieve these goals with a mechanism that (a) combines *local checkpoints* with *message replay*, thus avoiding both global checkpoint coordination and global rollbacks; (b) divides state of SEs into *consistent state*, which is checkpointed, and *dirty state*, which permits continued processing while checkpointing; and (c) partitions checkpoints and saves them to multiple nodes, which enables *parallel recovery*.

Approach. Our failure recovery mechanism combines local checkpointing and message logging and is inspired by failure recovery in distributed stream processing systems [14]. Nodes periodically take checkpoints of their local SEs and output communication buffers. Dataflows include increasing TE-generated scalar timestamps, and a vector timestamp of the last data item from each input dataflow that modified the SEs is included in the checkpoint. Once the checkpoint is saved to stable storage, upstream nodes can trim their output buffers of data items that are older than all downstream checkpoints.

After failure, a node recovers its SEs from the last checkpoint, replays its output buffers and reprocesses data items received from the upstream output buffers. Downstream nodes detect duplicate data items based on the timestamps and discard them. This approach allows nodes to recover SEs locally beyond the last checkpoint, without requiring nodes to coordinate global rollback, and it avoids the output commit problem.

State checkpointing. We use an asynchronous parallel checkpointing mechanism that minimises the processing interruption when checkpointing large SEs with GBs of memory. The idea is to record updates in a separate data structure, while taking a checkpoint. For each type of data structure held by an SE, there must be an implementation that supports the separation of dirty state and its subsequent consolidation.

Checkpointing of a node works as follows: (1) to initiate a checkpoint, each SE is flagged as *dirty* and the output buffers are added to the checkpoint; (2) updates from TEs to an SE are now handled using a *dirty state* data structure: e.g. updates to keys in a dictionary are written to the dirty state, and reads are first served by the dirty state and, only on a miss, by the dictionary; (3) asynchronously to the processing, the now consistent state is added to the checkpoint; (4) the checkpoint is backed up to multiple nodes (see below); and (5) the SE is locked and its state is consolidated with the dirty state.

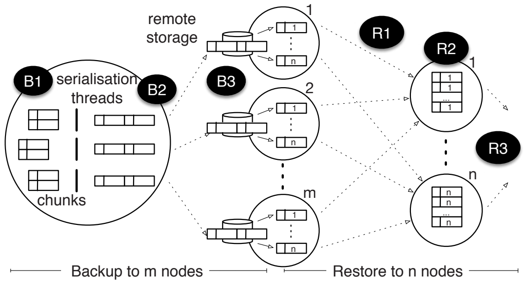


Figure 4: Parallel, *m*-to-*n* state backup and restore

State backup and restore. To be memory-efficient, checkpoints must be stored on disk. We overcome the problem of low I/O performance by splitting checkpoints across *m* nodes. To reduce recovery time, a failed SE instance can be restored to *n* new partitioned SE instances in parallel. This *m*-to-*n* pattern prevents a single node from becoming a disk, network or processing bottleneck.

Fig. 4 shows the distributed protocol for backing up checkpoints. In step B1, checkpoint chunks, e.g. obtained by hash-partitioning checkpoint data, are created, and a thread pool serialises them in parallel (step B2). Checkpoint chunks are streamed to *m* nodes, selected in a round-robin fashion (step B3). Nodes write received checkpoint chunks directly to disk.

After failure, *n* new nodes are instantiated with the lost TEs and SEs. Each node with a checkpoint chunk splits it into *n* partitions, each of which is streamed to one of the recovering instances (step R1). The new SE instances reconcile the chunks, reverting the partitioning (step R2). Finally, data items from output buffers are reprocessed to bring the recovered SE state up-to-date (step R3).

6 Evaluation

The goal of our experimental evaluation is to explore if SDGs can (i) execute stateful online processing applications with low latency and high throughput while supporting large state sizes with fine-grained updates (§6.1); (ii) scale in terms of nodes comparable to stateless batch processing frameworks (§6.2); handle stragglers at runtime with low impact on throughput (§6.3); and (iii) recover from failures with low overhead (§6.4).

We extend the SEEP streaming platform to implement SDGs and deploy our prototype on Amazon EC2 and a private cluster with 7 quad-core 3.4 GHz Intel Xeon servers with 8 GB of RAM. To support fast recovery, the checkpointing frequency for all experiments is 10 s unless stated otherwise. Candlesticks in plots show the 5th, 25th, 50th, 75th and 95th percentiles, respectively.

6.1 Stateful online processing

Throughput and latency. First we investigate the performance of SDGs using the *online collaborative filtering* (CF) application (see §2.1). We deploy it on 36 EC2 VM instances ("c1.xlarge"; 8 vCPUs with 7 GB) using the Netflix dataset, which contains 100 million movie

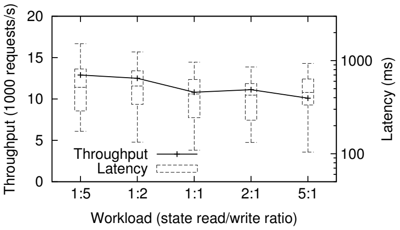


Figure 5: Throughput and latency with different read/write ratios (online collaborative filtering)

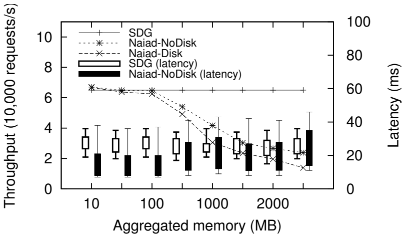


Figure 6: Throughput and latency with increasing state size on single node (key/value store)

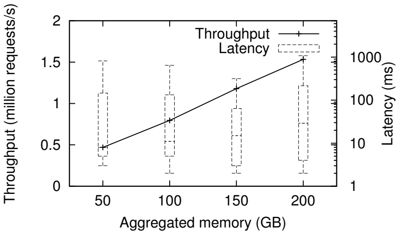


Figure 7: Throughput and latency with increasing state size on multiple nodes (key/value store)

ratings for evaluating recommender systems. We add new ratings continuously (addRating), while requesting fresh recommendations (getRec). The state size maintained by the system grows to 12 GB.

Fig. 5 shows the throughput of getRec and addRating requests and the latencies of getRec requests when the ratio between the two is varied. The achieved throughput is sufficient to serve 10,000–14,000 requests/s, with the 95th percentile of responses being at most 1.5 s stale. As the workload ratio includes more state reads (getRec), the throughput decreases slightly due to the cost of the synchronisation barrier that aggregates the partial state in the SDG. The result shows that SDGs can combine the functionality of a batch and an online processing system, while serving fresh results with low latency and high throughput over large mutable state.

State size. Next we evaluate the performance of SDGs as the state size increases. As a synthetic benchmark, we implement a distributed partitioned *key/value store* (KV) using SDGs because it exemplifies an algorithm with pure mutable state. We compare to an equivalent implementation in Naiad (version 0.2) with global checkpointing, which is the only fault-tolerance mechanism available in the open-source version. We deploy it in one VM (“m1.xlarge”) and measure the performance of serving update requests for keys.

Fig. 6 shows that, for a small state size of 100 MB, both SDGs and Naiad exhibit similar throughput of 65,000 requests/s with low latency. As the state size increases to 2.5 GB, the SDG throughput is largely unaffected but Naiad’s throughput decreases due to the overhead of its disk-based checkpoints (Naiad-Disk). Even with checkpoints stored on a RAM disk (Naiad-NoDisk), its throughput with 2.5 GB of state is 63% lower than that of SDGs. Similarly, the 95th percentile latency in Naiad increases when it stops processing during checkpointing—SDGs do not suffer from this problem.

To investigate how SDGs can support large distributed state across multiple nodes, we scale the KV store by increasing the number of VMs from 10 to 40, keeping the number of dictionary keys per node constant at 5 GB.

Fig. 7 shows the throughput and the latency for read requests with a given total state size. The aggregate throughput scales near linearly from 470,000 requests/s for 50 GB to 1.5 million requests/s for 200 GB. The median latency increases from 8–29 ms, while the 95th percentile latency varies between 800 ms and 1000 ms.

This result demonstrates that SDGs can support stateful applications with large state sizes without compromising throughput or processing latency, while executing in a fault-tolerant fashion.

Update granularity. We show the performance of SDGs when performing frequent, fine-grained updates to state. For this, we deploy a streaming *wordcount* (WC) application on 4 nodes in our private cluster. WC reports the word frequencies over a wall clock time window while processing the Wikipedia dataset. We compare to WC implementations in Streaming Spark [39] and Naiad.

We vary the size of the window, which controls the granularity at which input data updates the state: the smaller the window size, the less batching can be done when updating the state. Since Naiad permits the configuration of the batch size independently of the window size, we use a small batch size (1000 messages) for low-latency (Naiad-LowLatency) and a large one (20,000 messages) for high-throughput processing (Naiad-HighThroughput).

Fig. 8 shows that only SDG and Naiad-LowLatency can sustain processing for all window sizes, but SDG has a higher throughput due to Naiad’s scheduling overhead. The other deployments suffer from the overhead of micro-batching: Streaming Spark has a throughput similar to SDG, but its smallest sustainable window size is 250 ms, after which its throughput collapses; Naiad-HighThroughput achieves the highest throughput of all, but it also cannot support windows smaller than 100 ms. This shows that SDGs can perform fine-grained state updates without trading off throughput for latency.

6.2 Scalability

We explore if SDGs can scale to higher throughput with more nodes in a batch processing scenario. We deploy an implementation of *logistic regression* (LR) [21] on EC2 (“m1.xlarge”; 4 vCPUs with 15 GB). We compare to LR from Spark [38], which is designed for iterative processing, using the 100 GB dataset provided in its release.

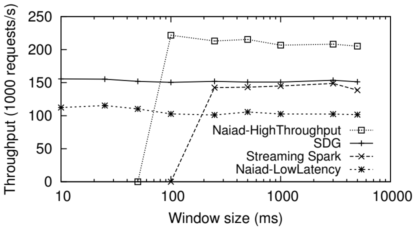


Figure 8: Latency with different window sizes (streaming wordcount)

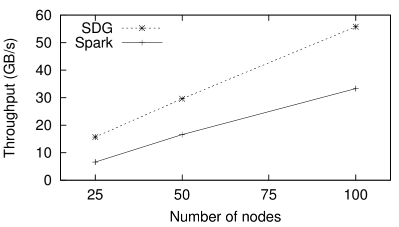


Figure 9: Scalability in terms of throughput (batch logistic regression)

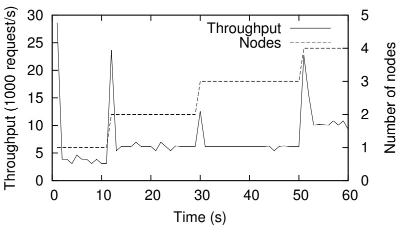


Figure 10: Runtime parallelism for handling stragglers (collaborative filtering)

Fig. 9 shows the throughput of our SDG implementation and Spark for 25–100 nodes. Both systems exhibit linear scalability. The throughput of SDGs is higher than Spark, which is likely due to the pipelining in SDGs, which avoids the re-instantiation of tasks after each iteration. With higher throughput, iterations are shorter, which leads to a faster convergence time. We conclude that the management of partial state in the LR application does not limit scalability compared to existing stateless dataflow systems.

6.3 Stragglers

We explore how SDGs handle straggling nodes by creating new TE and SE instances at runtime (see §3.3). For this, we deploy the CF application on our cluster and include a less powerful machine (2.4 GHz with 4 GB).

Fig. 10 shows how the throughput and the number of nodes changes over time as bottleneck TEs are identified by the system. At the start, a single instance of the `getRecVec` TE is deployed. It is identified as a bottleneck, and a second instance is added at $t = 10$ s, which also causes a new instance of the partial state in the `coOcc` matrix to be created. This increases the throughput from 3600–6200 requests/s. The throughput spikes occur when the input queues of new TE instances fill up.

Since the new node is allocated on the less powerful machine, it becomes a straggler, limiting overall throughput. At $t = 30$ s, adding a new TE instance without relieving the straggler does not increase the throughput. At $t = 50$ s, the straggling node is detected by the system, and a new instance is created to share its work. This increases the throughput from 6200–11,000 requests/s.

This shows how straggling nodes are mitigated by allocating new TE instances on-demand, distributing new partial or partitioned SE instances as required. In more extreme cases, a straggling node could even be removed and the job resumed from a checkpoint with new nodes.

6.4 Failure recovery

We evaluate the performance and overhead of our failure recovery mechanism for SDGs. We (i) explore the recovery time under different recovery strategies; (ii) assess the advantages of our asynchronous checkpointing mechanism; and (iii) investigate the overhead with different checkpointing frequencies and state sizes. We deploy the KV store on one node of our cluster, together with spare nodes to store backups and replace failed nodes.

Recovery time. We fail the node under different recovery strategies: an *m*-to-*n* recovery strategy uses *m* backup nodes to restore to *n* recovered nodes (see §5). For each, we measure the time to restore the lost SE, re-process unprocessed data and resume processing.

Fig. 11 shows the recovery times for different SE sizes under different strategies: (i) the simplest strategy, 1-to-1, has the longest recovery time, especially with large state sizes, because the state is restored from a single node; (ii) the 2-to-1 strategy streams checkpoint chunks from two nodes in parallel, which improves disk I/O throughput but also increases the load on the recovering node when it reconstitutes the state; (iii) in the 1-to-2 strategy, checkpoint chunks are streamed to two recovering nodes, thus halving the load of state reconstruction; and (iv) the 2-to-2 strategy recovers fastest because it combines the above two strategies—it parallelises both the disk reads and the state reconstruction.

As the state becomes large, state reconstruction dominates over disk I/O overhead: with 4 GB, streaming from two disks does not improve recovery time. Adopting a strategy that recovers a failed node with multiple nodes, however, has significant benefit, compared to cases with smaller state sizes.

Synchronous vs. asynchronous checkpointing. We investigate the benefit of our asynchronous checkpointing mechanism in comparison with synchronous checkpointing that stops processing, as used by Naiad [26] and SEEP [10].

Fig. 12 compares the throughput and 99th percentile latency with increasing state sizes. As the checkpoint size grows from 1–4 GB, the average throughput under synchronous checkpointing reduces by 33%, and the latency increases from 2–8 s because the system stops processing while checkpointing. With asynchronous checkpointing, there is only a small (~5%) impact on throughput. Latency is an order of magnitude lower and only moderately affected (from 200–500 ms). This result shows that a synchronous checkpointing approach cannot achieve low-latency processing with large state sizes.

Overhead of asynchronous checkpointing. Next we evaluate the overhead of our checkpointing mechanism

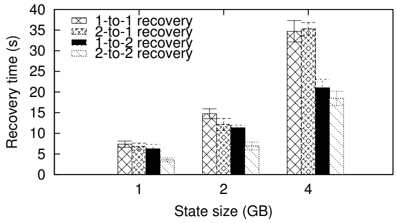


Figure 11: Recovery times with different *m*-to-*n* recovery strategies

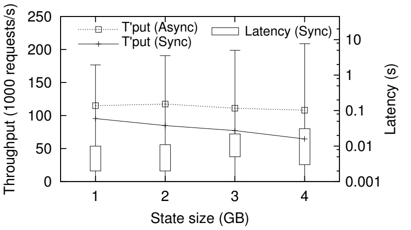


Figure 12: Comparison of sync. and async. checkpointing

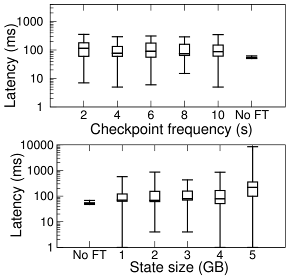


Figure 13: Impact of checkpoint frequency and size on latency

as a function of checkpointing frequency and state size.

Fig. 13 (top) shows the processing latency when varying the checkpointing frequency. The rightmost data point (No FT) represents the case where the checkpointing mechanism is disabled. The bottom figure reports the impact of the size of the checkpoint on latency.

Checkpointing has a limited impact on latency: without fault tolerance, the 95th percentile latency is 68 ms, and it increases to 500 ms when checkpointing 1 GB every 10 s. This is due to the overhead of merging dirty state and saving checkpoints to disk. Increasing the checkpointing frequency or size gradually also increases latency: the 95th percentile latency with 4 GB is 850 ms, while checkpointing 2 GB every 4 s results in 1 s.

Beyond that, the checkpointing overhead starts to impact higher percentiles more significantly. Checkpointing frequency and size behave almost proportionally: as the state size increases, the frequency can be reduced to maintain a low processing latency.

Overall this experiment demonstrates the strength of our checkpointing mechanism, which only locks state while merging dirty state. The locking overhead thus reduces proportionally to the state update rate.

7 Related Work

Programming model. Data-parallel frameworks typically support a functional/declarative model: MapReduce [8] only has two higher-order functions; more recent frameworks [15, 38, 13] permit user-defined functional operators; and Naiad [26] supports different functional and declarative programming models on top of its timely dataflow model. CBP [19], Storm and SEEP [10] expose a low-level dataflow programming model: algorithms are defined as a dataflow pipeline, which is harder to program and debug. While functional and dataflow models ease distribution and fault tolerance, SDGs target an imperative programming model, which remains widely used by data scientists [17].

Efforts exist to bring imperative programming to data-parallel processing. CIEL [25] uses imperative constructs such as task spawning and futures, but this exposes the low-level execution of the dynamic dataflow graph to developers. Piccolo [30] and Oolong [24] offer imperative compute kernels with distributed state, which requires algorithms to be structured accordingly.

In contrast, SDGs simplify the translation of imperative programs to dataflows using basic program analysis techniques, which infer state accesses and the dataflow. By separating different types of state access, it becomes possible to choose automatically an effective implementation for distributed state.

GraphLab [20] and Pregel [22] are frameworks for graph computations based on a shared-memory abstraction. They expose a vertex-centric programming model whereas SDGs target generic stateful computation.

Program parallelisation. Matlab has language constructs for parallel processing of large datasets on clusters. However, it only supports the parallelisation of sequential blocks or iterations and not of general dataflows.

Declarative models such as Pig [28], DyradLINQ [37], SCOPE [6] and Stratosphere [9] are naturally amenable to automatic parallelisation—functions are stateless, which allows data-parallel versions to execute on multiple nodes. Instead, we focus on an imperative model.

Other approaches offer new programming abstractions for parallel computation over distributed state. FlumeJava [7] provides distributed immutable collections. While immutability simplifies parallel execution, it limits the expression of imperative algorithms. In Piccolo [30], global mutable state is accessed remotely by parallel distributed functions. In contrast, tasks in SDGs only access local state with low latency, and state is always colocated with computation. Presto [35] has distributed partitioned arrays for the R language. Partitions can be collected but not updated by multiple tasks, whereas SDGs permit arbitrary dataflows.

Extracting parallel dataflows from imperative programs is a hard problem [16]. We follow an approach similar to that of Beck et al. [3], in which a dataflow graph is generated compositionally from the execution graph. While early work focused on hardware-based dataflow models [27], more recent efforts target thread-based execution [18]. Our problem is simpler because we do not extract task parallelism but only focus on data and pipeline parallelism in relation to distributed state access.

Similar to pragma-based techniques [34], we use annotations to transform access to distributed state into access to local instances. Blazes [2] uses annotations to generate automatically coordination code for distributed programs. Our goal is different: SDGs execute imperative code in a distributed fashion, and coordination is determined by the extracted dataflow.

Failure recovery. In-memory systems are prone to failures [1], and fast recovery is important for low-latency and high-throughput processing. With large state sizes, checkpoints cannot be stored in memory, but storing them on disk can increase recovery time. RAMCloud [29] replicates data across cluster memory and eventually backs it up to persistent storage. Similar to our approach, data is recovered from multiple disks in parallel. However, rather than replicating each write request, we checkpoint large state atomically, while permitting new requests to operate on dirty state.

Streaming Spark [39] and Spark [38] use RDDs for recovery. After a failure, RDDs are recomputed in parallel on multiple nodes. Such a recovery mechanism is effective if recomputation is inexpensive—for state that depends on the entire history of the data, it would be prohibitive. In contrast, the parallel recovery in SDGs retrieves partitioned checkpoints from multiple nodes, and only reprocesses data from output buffers to bring restored SE instances up-to-date.

8 Conclusions

Data-parallel processing frameworks must offer a familiar programming model with good performance. Supporting imperative online machine learning algorithms poses challenges to frameworks due to their use of large distributed state with fine-grained access.

We describe stateful dataflow graphs (SDGs), a data-parallel model that is designed to offer a dataflow abstraction over large mutable state. With the help of annotations, imperative algorithms can be translated to SDGs, which manage partitioned or partial distributed state. As we demonstrated in our evaluation, SDGs can support diverse stateful applications, thus generalising a number of existing data-parallel computation models.

Acknowledgements. This work was supported by a PhD CASE Award funded by EPSRC/BAE Systems. We thank our PC contact, Jinyang Li, and the anonymous ATC reviewers for their feedback and guidance.

References

[1] Akidau, T., Balikov, A., et al. MillWheel: Fault-Tolerant Stream Processing at Internet Scale. In *VLDB* (2013).

[2] Alvaro, P., Conway, N., et al. Blazes: Coordination Analysis for Distributed Programs. In *ICDE* (2014).

[3] Beck, M., and Pingali, K. From Control Flow to Dataflow. In *ICPP* (1990).

[4] Bhatotia, P., Wieder, A., et al. Incoop: MapReduce for Incremental Computations. In *SOCC* (2011).

[5] Bu, Y., Howe, B., et al. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB* (2010).

[6] Chaiken, R., Jenkins, B., et al. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *VLDB* (2008).

[7] Chambers, C., Raniwala, A., et al. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *PLDI* (2010).

[8] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. In *CACM* (2008).

[9] Ewen, S., Tzoumas, K., et al. Spinning Fast Iterative Data Flows. In *VLDB* (2012).

[10] Fernandez, R. C., Migliavacca, M., et al. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD* (2013).

[11] Gunda, P. K., Ravindranath, L., et al. Nectar: Automatic Management of Data and Comp. in Datacenters. In *OSDI* (2010).

[12] He, B., Yang, M., et al. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *SOCC* (2010).

[13] Hueske, F., Peters, M., et al. Opening the Black Boxes in Data Flow Optimization. In *VLDB* (2012).

[14] Hwang, J.-H., Balazinska, M., et al. High-Availability Algorithms for Distributed Stream Processing. In *ICDE* (2005).

[15] Isard, M., Budiu, M., et al. Dryad: Dist. Data-Parallel Programs from Sequential Building Blocks. In *EuroSys* (2007).

[16] Johnston, W. M., Hanna, J., et al. Advances in Dataflow Programming Languages. In *CSUR* (2004).

[17] KDnuggets Annual Software Poll. RapidMiner and R vie for the First Place. `http://goo.gl/OLikb`, 2013.

[18] Li, F., Pop, A., et al. Automatic Extraction of Coarse-Grained Data-Flow Threads from Imperative Programs. In *Micro* (2012).

[19] Logothethis, D., Olson, C., et al. Stateful Bulk Processing for Incremental Analytics. In *SOCC* (2010).

[20] Low, Y., Bickson, D., et al. Dist. GraphLab: A Framework for ML and Data Mining in the Cloud. In *VLDB* (2012).

[21] Ma, J., Saul, L. K., et al. Identifying Suspicious URLs: an Application of Large-Scale Online Learning. In *ICML* (2009).

[22] Malewicz, G., Austern, M. H., et al. Pregel: A System for Large-scale Graph Processing. In *SIGMOD* (2010).

[23] Mishne, G., Dalton, J., et al. Fast Data in the Era of Big Data: Twitter's Real-Time Related Query Suggestion Architecture. In *SIGMOD* (2013).

[24] Mitchell, C., Power, R., et al. Oolong: Asynchronous Distributed Applications Made Easy. In *APSYS* (2012).

[25] Murray, D., Schwarzkopf, M., et al. CIEL: A Universal Exec. Engine for Distributed Data-Flow Comp. In *NSDI* (2011).

[26] Murray, D. G., McSherry, F., et al. Naiad: A Timely Dataflow System. In *SOSP* (2013).

[27] Nikhil, R. S., et al. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *TC* (1990).

[28] Olston, C., Reed, B., et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD* (2008).

[29] Ongaro, D., Rumble, S. M., et al. Fast Crash Recovery in RAMcloud. In *SOSP* (2011).

[30] Power, R., and Li, J. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *OSDI* (2010).

[31] Schelter, S., Ewen, S., et al. All Roads Lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. In *CIKM* (2013).

[32] Sumbaly, R., Kreps, J., et al. The Big Data Ecosystem at LinkedIn. In *SIGMOD* (2013).

[33] Vallée-Rai, R., Hendren, L., et al. Soot: A Java Optimization Framework. In *CASCON* (1999).

[34] Vandierendonck, H., Rul, S., et al. The Paralax Infrastructure: Automatic Parallelization with a Helping Hand. In *PACT* (2010).

[35] Venkataraman, S., Bodzsar, E., et al. Presto: Dist. ML and Graph Processing with Sparse Matrices. In *EuroSys* (2013).

[36] Xin, R. S., Rosen, J., et al. Shark: SQL and Rich Analytics at Scale. In *SIGMOD* (2013).

[37] Yu, Y., Isard, M., et al. DryadLINQ: a System for General-Purpose Distributed Data-Parallel Computing using a High-Level Language. In *OSDI* (2008).

[38] Zaharia, M., Chowdhury, M., et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI* (2012).

[39] Zaharia, M., Das, T., et al. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *SOSP* (2013).

[40] Zaharia, M., Konwinski, A., et al. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI* (2008).

OS^v— Optimizing the Operating System for Virtual Machines

Avi Kivity Dor Laor Glauber Costa Pekka Enberg
Nadav Har'El Don Marti Vlad Zolotarov
Cloudius Systems
{avi,dor,glommer,penberg,nyh,dmarti,vladz}@cloudius-systems.com

Abstract

Virtual machines in the cloud typically run existing general-purpose operating systems such as Linux. We notice that the cloud's hypervisor already provides some features, such as isolation and hardware abstraction, which are duplicated by traditional operating systems, and that this duplication comes at a cost.

We present the design and implementation of OS^v, a new guest operating system designed specifically for running a single application on a virtual machine in the cloud. It addresses the duplication issues by using a low-overhead library-OS-like design. It runs existing applications written for Linux, as well as new applications written for OS^v. We demonstrate that OS^v is able to efficiently run a variety of existing applications. We demonstrate its sub-second boot time, small OS image and how it makes more memory available to the application. For unmodified network-intensive applications, we demonstrate up to 25% increase in throughput and 47% decrease in latency. By using non-POSIX network APIs, we can further improve performance and demonstrate a 290% increase in Memcached throughput.

1 Introduction

Cloud computing (Infrastructure-as-a-Service, or IaaS) was born out of the realization that virtualization makes it easy and safe for different organizations to share one pool of physical machines. At any time, each organization can rent only as many virtual machines as it currently needs to run its application.

Today, virtual machines on the cloud typically run the same traditional operating systems that were used on physical machines, e.g., Linux, Windows, and *BSD. But as the IaaS cloud becomes ubiquitous, this choice is starting to make less sense: The features that made these operating systems desirable on physical machines, such as familiar single-machine administration interfaces and support for a large selection of hardware, are losing their relevance. At the same time, different features are becoming important: The VM's operating system needs to be fast, small, and easy to administer at large scale.

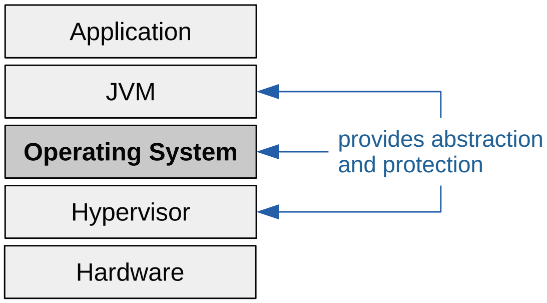


Figure 1: Software layers in a typical cloud VM.

Moreover, fundamental features of traditional operating systems are becoming overhead, as they are now duplicated by other layers of the cloud stack (illustrated in Figure 1).

For example, an important role of traditional operating systems is to isolate different processes from one another, and all of them from the kernel. This isolation comes at a cost, in performance of system calls and context switches, and in complexity of the OS. This was necessary when different users and applications ran on the same OS, but on the cloud, the hypervisor provides isolation between different VMs so mutually-untrusting applications do not need to run on the same VM. Indeed, the scale-out nature of cloud applications already resulted in a trend of focused single-application VMs.

A second example of duplication is hardware abstraction: An OS normally provides an abstraction layer through which the application accesses the hardware. But on the cloud, this "hardware" is itself a virtualized abstraction created by the hypervisor. Again, this duplication comes at a performance cost.

This paper explores the question of what an operating system would look like if we designed it today with the sole purpose of running on virtual machines on the cloud, and not on physical machines.

We present OS^v, a new OS we designed specifically for cloud VMs. The main goals of OS^v are as follows:

- Run existing cloud applications (Linux executables).
- Run these applications faster than Linux does.
- Make the image small enough, and the boot quick enough, that starting a new VM becomes a viable alternative to reconfiguring a running one.
- Explore new APIs for new applications written for OS^v, that provide even better performance.
- Explore using such new APIs in common runtime environments, such as the Java Virtual Machine (JVM). This will boost the performance of unmodified Java applications running on OSv.
- Be a platform for continued research on VM operating systems. OS^v is actively developed as open source, it is written in a modern language (C++11), its codebase is relatively small, and our community encourages experimentation and innovation.

OS^v supports different hypervisors and processors, with only minimal amount of architecture-specific code. For 64-bit x86 processors, it currently runs on the KVM, Xen, VMware and VirtualBox hypervisors, and also on the Amazon EC2 and Google GCE clouds (which use a variant of Xen and KVM, respectively). Preliminary support for 64-bit ARM processors is also available.

In Section 2, we present the design and implementation of OS^v. We will show that OS^v runs only on a hypervisor, and is well-tuned for this (e.g., by avoiding spinlocks). OS^v runs a single application, with the kernel and multiple threads all sharing a single address space. This makes system calls as efficient as function calls, and context switches quicker. OS^v supports SMP VMs, and has a redesigned network stack (*network channels*) to lower socket API overheads. OS^v includes other facilities one expects in an operating system, such as standard libraries, memory management and a thread scheduler, and we will briefly survey those. OS^v's scheduler incorporates several new ideas including lock-free algorithms and floating-point based fair accounting of run-time.

In Section 3, we begin to explore what kind of new APIs a single-application OS like OS^v might have beyond the traditional POSIX APIs to further improve performance. We suggest two techniques to improve JVM memory utilization and garbage-collection performance, which boost performance of all JVM languages (Java, Scala, Jruby, etc.) on OS^v. We then demonstrate that a zero-copy, lock-free API for packet processing can result in a 4x increase of Memcached throughput.

In Section 4, we evaluate our implementation, and compare OS^v to Linux on several micro- and macro-benchmarks. We show minor speedups over Linux in computation- and memory-intensive workloads such as the SPECjvm2008 benchmark, and up to 25% increase in throughput and 47% reduction in latency in network-dominated workloads such as Netperf and Memcached.

2 Design and Implementation of OS^v

OS^v follows the *library OS* design, an OS construct pioneered by *exokernels* in the 1990s [5]. In OS^v's case, the hypervisor takes on the role of the exokernel, and VMs the role of the applications: Each VM is a single application with its own copy of the library OS (OS^v). Library OS design attempts to address performance and functionality limitations in applications that are caused by traditional OS abstractions. It moves resource management to the application level, exports hardware directly to the application via safe APIs, and reduces abstraction and protection layers.

OS^v runs a single application in the VM. If several mutually-untrusting applications are to be run, they can be run in separate VMs. Our assumption of a single application per VM simplifies OS^v, but more importantly, eliminates the redundant and costly isolation inside a guest, leaving the hypervisor to do isolation. Consequently, OS^v uses a *single address space* — all threads and the kernel use the same page tables, reducing the cost of context switches between applications threads or between an application thread and the kernel.

The OS^v kernel includes an ELF dynamic linker which runs the desired application. This linker accepts standard ELF dynamically-linked code compiled for Linux. When this code calls functions from the Linux ABI (i.e., functions provided on Linux by the glibc library), these calls are resolved by the dynamic linker to functions implemented by the OS^v kernel. Even functions which are considered "system calls" on Linux, e.g., `read()`, in OS^v are ordinary function calls and do not incur special call overheads, nor do they incur the cost of user-to-kernel parameter copying which is unnecessary in our single-application OS.

Aiming at compatibility with a wide range of existing applications, OS^v emulates a big part of the Linux programming interface. Some functions like `fork()` and `exec()` are not provided, since they don't have any meaning in the one-application model employed by OS^v.

The core of OS^v is new code, written in C++11. This includes OS^v's loader and dynamic linker, memory management, thread scheduler and synchronization mechanisms such as mutex and RCU, virtual-hardware drivers, and more. We will discuss below some of these mechanisms in more detail.

Operating systems designed for physical machines usually devote much of their code to supporting diverse hardware. The situation is much easier for an operating system designed for VMs, such as OS^v, because hypervisors export a simplified and more stable hardware view. OS^v has drivers for a small set of traditional PC devices commonly emulated by hypervisors, such as a keyboard, VGA, serial port, SATA, IDE and HPET. Additionally, it supports several paravirtual drivers for improved performance: A paravirtual clock is supported on KVM and Xen, a paravirtual NIC using virtio [25] and VMXNET3 [29], and a paravirtual block device (disk) using virtio and pvscsi.

For its filesystem support, OS^v follows a traditional UNIX-like VFS (virtual filesystem) design [12] and adopts ZFS as its major filesystem. ZFS is a modern filesystem emphasizing data integrity and advanced features such as snapshots and volume management. It employs a modified version of the Adaptative Replacement Cache [18] for page cache management and consequently it can achieve a good balance between recency and frequency hits.

Other filesystems are also present in OS^v. There is one in-memory filesystem for specialized applications that may want to boot without disk (ramfs), and a very simple device filesystem for device views (devfs). For compatibility with Linux applications, a simplified procfs is also supported.

Some components of OS^v were not designed from scratch, but rather imported from other open-source projects. We took the C library headers and some functions (such as stdio and math functions) from the *musl libc* project, the VFS layer from *Prex* project, the ZFS filesystem from *FreeBSD*, and and the ACPI drivers from the *ACPICA* project. All of these are areas in which OS^v's core value is not expected to be readily apparent so it would make less sense for these to be written from scratch, and we were able to save significant time by reusing existing implementations.

OS^v's network stack was also initially imported from FreeBSD, because it was easier to start with an implementation known to be correct, and later optimize it. As we explain in Section 2.3, after the initial import we rewrote the network stack extensively to use a more efficient *network channels*-based design.

It is beyond the scope of this article to cover every detail of OS^v's implementation. Therefore, the remainder of this section will explore a number of particularly interesting or unique features of OS^v's implementation, including: 1. memory management in OS^v, 2. how and why OS^v completely avoids spinlocks, 3. network channels, a non-traditional design for the networking stack, and 4. the OS^v thread scheduler, which incorporates several new ideas including lock-free algorithms and floating-point based fair accounting of run-time.

2.1 Memory Management

In theory, a library OS could dictate a flat physical memory mapping. However, OS^v uses virtual memory like general purpose OSs. There are two main reasons for this. First, the x86_64 architecture mandates virtual memory usage for long mode operation. Second, modern applications following traditional POSIX-like APIs tend to map and unmap memory and use page protection themselves.

OS^v supports demand paging and memory mapping via the `mmap` API. This is important, for example, for a class of JVM-based applications that bypass the JVM and use `mmap` directly through JNI. Such applications include Apache Cassandra which is a popular NoSQL database running on the JVM.

For large enough mappings, OS^v will fill the mapping with huge pages (2MB in size for the x86_64 architecture). The use of larger page sizes improve performance of applications by reducing the number of TLB misses. [24].

Since mappings can be partially unmapped, it is possible that one of these pages needs to be broken into smaller pages. By employing a mechanism similar to Linux's Transparent Huge Pages, OS^v handles this case transparently.

As an OS that aims to support a single application, page eviction is not supported. Additional specialized memory management constructs are described in Section 3.

2.2 No Spinlocks

One of the primitives used by contemporary OSs on SMP machines is the spin-lock [2]. On a *single*-processor system, it is easy to protect a data structure from concurrent access by several contexts by disabling interrupts or context switches while performing non-atomic modifications. That is not enough on *multi*-processor systems, where code running on multiple CPUs may touch the data concurrently. Virtually all modern SMP OSs today use spin-locks: One CPU acquires the lock with an atomic test-and-set operation, and the others execute a busy-loop until they can acquire the lock themselves. SMP OSs use this spin-lock primitive to implement higher-level locking facilities such as *sleeping mutexes*, and also use spin-locks directly in situations where sleeping is forbidden, such as in the scheduler itself and in interrupt-handling context.

Spin-locks are well-suited to a wide range of SMP physical hardware. However when we consider *virtual* machines, spin-locks suffer from a significant drawback known as the "lock-holder preemption" problem [28]: while physical CPUs are always running if the OS wants them to, virtual CPUs may "pause" at unknown times for unknown durations. This can happen during exits to the hypervisor or because the hypervisor decides to run other guests or even hypervisor processes on this CPU.

If a virtual CPU is paused while holding a spin-lock, other CPUs that want the same lock spin needlessly, wasting CPU time. When a mutex is implemented using a spin-lock, this means that a thread waiting on a lock can find itself spinning and wasting CPU time, instead of immediately going to sleep and letting another thread run. The consequence of the lock-holder preemption problem is lower performance — Friebel et al. have shown that multitasking two guests on the same CPU results in performance drops from 7% up to 99% in extreme cases [7].

Several approaches have been proposed to mitigate the lock-holder preemption problem [7], usually requiring changes to the hypervisor or some form of cooperation between the hypervisor and the guest. However, in a kernel designed especially to run in a virtual machine, a better solution is to avoid the problem completely. OS^v does not use spin-locks *at all*, without giving up on lock-based algorithms in the kernel or restricting it to single-processor environments.

One way to eliminate spin-locks is to use lock-free algorithms [19]. These algorithms make clever use of various atomic instructions provided by the SMP machine (e.g., *compare-exchange*, *fetch-and-add*) to ensure that a data structure remains in consistent state despite concurrent modifications. We can also avoid locks by using other techniques such as *Read-Copy-Update* (RCU) [17]. But lock-free algorithms are very hard to develop, and it is difficult to completely avoid locking in the kernel [16], especially considering that we wanted to re-use existing kernel components such as ZFS and the BSD network stack. Therefore, our approach is as follows:

1. Ensure that most work in the kernel, including interrupt handling, is done in threads. These can use lock-based algorithms: They use a mutex (which can put a thread to sleep), not a spin-lock.

2. Implement the mutex itself without using a spin-lock, i.e., it is a lock-free algorithm.

3. The scheduler itself cannot be run in a thread, so to protect its data structures without spin-locks, we use per-cpu run queues and lock-free algorithms.

OS^v executes almost everything in ordinary threads. Interrupt handlers usually do nothing but wake up a thread which will service the interrupting device. Kernel code runs in threads just like application code, and can sleep or be preempted just the same. OS^v's emphasis on cheap thread context switches ensures that the performance of this design does not suffer.

Our mutex implementation is based on a lock-free design by Gidenstam & Papatriantafilou [8], which protects the mutex's internal data structures with atomic operations in a lock-free fashion. With our lock-free mutex, a paused CPU cannot cause other CPUs to start spinning. As a result, kernel and application code which uses this mutex are free from the lock-holder preemption problem.

Finally, the scheduler itself uses per-CPU run queues, so that most scheduling decisions are local to the CPU and need no locking. It uses lock-free algorithms when scheduling cooperation is needed across CPUs, such as waking a thread that belongs to a different CPU. OS^v's scheduler is described in more detail in Section 2.4.

2.3 Network Channels

An operating system designed for the cloud must, almost by definition, provide a high quality TCP/IP stack. OS^v does this by applying Van Jacobson's *net channel* ideas [10] to its networking stack.

We begin by observing that a typical network stack is traversed in two different directions:

- Top-down: the `send()` and `recv()` system calls start at the socket layer, convert user buffers to TCP packets, attach IP headers to those TCP packets, and finally egress via the network card driver,

- Bottom-up: incoming packets are received by the network card driver, parsed by the IP layer, forwarded to the TCP layer, and are then appended to socket buffers; blocked `send()`, `recv()`, and `poll()` system calls are then woken as necessary.

As illustrated in Figure 2a, both the interrupt contexts (hard- and soft- interrupt) and the application thread context perform processing on all layers of the network stack. The key issue is that code from both contexts accesses shared data structures, causing lock and cache-line contention on heavily used connections.

In order to resolve this contention, under OS^v almost all packet processing is performed in an application thread. Upon packet receipt, a simple classifier associates it with a *channel*, which is a single producer/single consumer queue for transferring packets to the application thread. Each channel corresponds to a single flow, such as a TCP connection or a UDP path from an interface to a socket.

As can be seen in Figure 2b, access to shared data structures from multiple threads is completely eliminated (save for the channel itself).

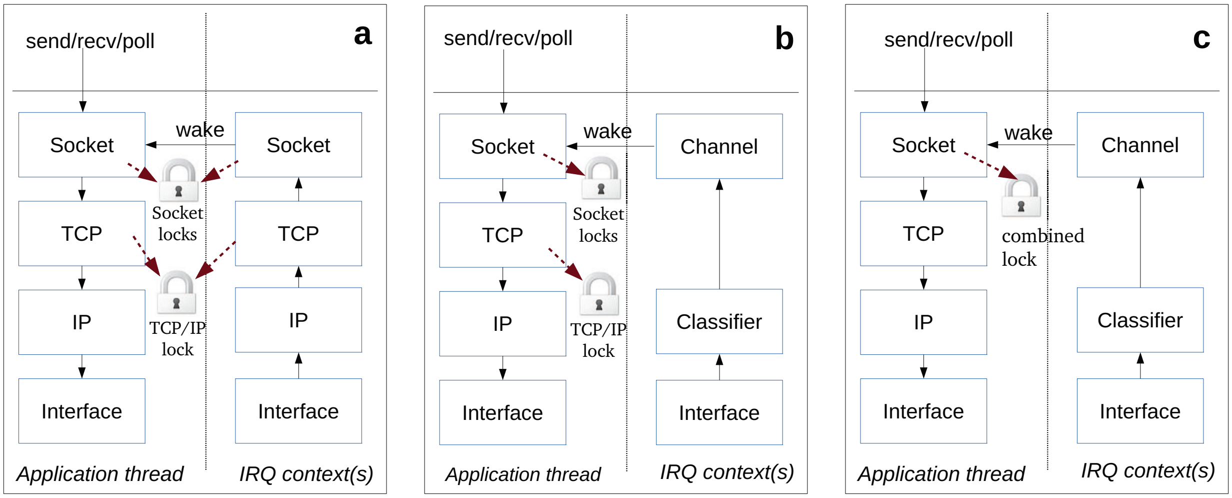


Figure 2: Control flow and locking in (left to right): (a) a traditional networking stack, (b) OS^v's networking stack prior to lock merging, and (c) OS^v's complete networking stack

In addition, since there is now just one thread accessing the data, locking can be considerably simplified, reducing both run-time and maintenance overhead.

Switching to a net channel approach allows a significant reduction in the number of locks required, leading to the situation in Figure 2c:

- The socket receive buffer lock has been merged with the socket send buffer lock; since both buffers are now populated by the same thread (running either the `send()` or `recv()` system calls), splitting that lock is unnecessary,
- The interleave prevention lock (used to prevent concurrent writes from interleaving) has been eliminated and replaced by a wait queue using the socket lock for synchronization, and
- The TCP layer lock has been merged with the socket layer lock; since TCP processing now always happens within the context of a socket call, it is already protected by that lock.

We expect further simplifications and improvements to the stack as it matures.

2.4 The Thread Scheduler

The guiding principles of OS^v's thread scheduler are that it should be *lock-free*, *preemptive*, *tick-less*, *fair*, *scalable* and *efficient*.

Lock-free As explained in Section 2.2, OS^v's scheduler should not use spin-locks and it obviously cannot use a sleeping mutex.

The scheduler keeps a separate *run queue* on each CPU, listing the *runnable* threads on the CPU. Sleeping threads are not listed on any run queue. The scheduler runs on a CPU when the running thread asks for a reschedule, or when a timer expiration forces preemption. At that point, the scheduler chooses the most appropriate thread to run next from the threads on this CPU's run-queue, according to its fairness criteria. Because each CPU has its own separate run-queue, this part of the scheduler needs no locking.

The separate run queues can obviously lead to a situation where one CPU's queue has more runnable threads than another CPU's, hurting the scheduler's overall fairness. We solve this by running a *load balancer* thread on each CPU. This thread wakes up once in a while (10 times a second), and checks if some other CPU's run queue is shorter than this CPU's. If it is, it picks one thread from this CPU's run queue, and wakes it on the remote CPU.

Waking a thread on a remote CPU requires a more elaborate lock-free algorithm: For each of the N CPUs, we keep N lock-free queues of *incoming wakeups*, for a total of N^2 queues. We also keep a bitmask of nonempty queues for each CPU. When CPU s wants to wake a thread on CPU d, it adds this thread to the queue (s,d), atomically turns on bit s in CPU d's bitmask and sends an inter-processor interrupt (IPI) to CPU d. The interrupt leads CPU d to perform a reschedule, which begins by looking for incoming wakeups. The bitmask tells the scheduler which of the incoming queues it needs to inspect.

Preemptive OS^v fully supports preemptive multitasking: While threads can voluntarily cause a reschedule (by waiting, yielding, or waking up another thread), one can also happen at any time, preempted by an interrupt such as a timer or the wakeup IPI mentioned above. All threads are preemptable and as with the rest of the system, there is no difference between application and kernel threads. A thread can temporarily avoid being preempted by incrementing a per-thread preempt-disable counter. This feature can be useful in a number of cases including, for example, maintaining per-CPU variables and RCU [17] locks. An interrupt while the running thread has preemption disabled will not cause a reschedule, but when the thread finally re-enables preemption, a reschedule will take place.

Tick-less Most classic kernels, and even many modern kernels, employ a periodic timer interrupt, also known as a *tick*. The tick causes a reschedule to happen periodically, for example, 100 times each second. Such kernels often account the amount of time that each thread has run in whole ticks, and use these counts to decide which thread to schedule at each tick.

Ticks are convenient, but also have various disadvantages. Most importantly, excessive timer interrupts waste CPU time. This is especially true on on virtual machines where interrupts are significantly slower than on physical machines, as they involve exits to the hypervisor.

Because of the disadvantages of ticks, OS^v implements a tickless design. Using a high resolution clock, the scheduler accounts to each thread the exact time it consumed, instead of approximating it with ticks. Some timer interrupts are still used: Whenever the fair scheduling algorithm decides to run one thread, it also calculates when it will want to switch to the next thread, and sets a timer for that period. The scheduler employs hysteresis to avoid switching too frequently between two busy threads. With the default hysteresis setting of 2ms, two busy threads with equal priority will alternate 4ms time slices, and the scheduler will never cause more than 500 timer interrupts each second. This number will be much lower when there aren't several threads constantly competing for CPU.

Fair On each reschedule, the scheduler must decide which of the CPU's runnable threads should run next, and for how long. A fair scheduler should account for the amount of *run time* that each thread got, and strive to either equalize it or achieve a desired ratio if the threads have different priorities. However, using the total runtime of the threads will quickly lead to imbalances. For instance, if a thread was out of the CPU for 10 seconds and becomes runnable, it will monopolize the CPU for 10 whole seconds as the scheduler seeks to achieve fairness. Instead, we want to equalize the amount of run-time that runnable threads have gotten in recent history, and forget about the distant past.

OS^v's scheduler calculates the exponentially-decaying moving average of each thread's recent run time. The scheduler will choose to run next the runnable thread with the lowest moving-average runtime, and calculate exactly how much time this thread should be allowed to run before its runtime surpasses that of the runner-up thread.

Our moving-average runtime is a floating-point number. It is interesting to mention that while some kernels forbid floating-point use inside the kernel, OS^v fully allows it. As a matter of fact, it has no choice but to allow floating point in the kernel due to the lack of a clear boundary between the kernel and the application.

The biggest stumbling block to implementing moving-average runtime as described above is its scalability: It would be impractical to update the moving-average runtimes of *all* threads on each scheduler invocation.

But we can show that this is not actually necessary; we can achieve the same goal with just updating the runtime of the single running thread. It is beyond the scope of this article to derive the formulas used in OS^v's scheduler to maintain the moving-average runtime, or to calculate how much time we should allow a thread to run until its moving-average runtime overtakes that of the runner-up thread.

Scalable OS^v's scheduler has $O(\lg N)$ complexity in the number of runnable threads on each CPU: The run queue is kept sorted by moving-average runtime, and as explained, each reschedule updates the runtime of just one thread. The scheduler is totally unaware of threads which are not runnable (e.g., waiting for a timer or a mutex), so there is no performance cost in having many utility threads lying around and rarely running. OS^v indeed has many of these utility threads, such as the load-balancer and interrupt-handling threads.

Efficient Beyond the scheduler's scalability, OS^v employs additional techniques to make the scheduler and context switches more efficient.

Some of these techniques include:

- OS^v's single address space means that we do not need to switch page tables or flush the TLB on context switches. This makes context switches significantly cheaper than those on traditional multi-process operating systems.

- Saving the floating-point unit (FPU) registers on every context switch is also costly. We make use of the fact that most reschedules are voluntary, caused by the running thread calling a function such as `mutex_wait()` or `wake()`. The x86_64 ABI guarantees that the FPU registers are caller-saved. So for voluntary context switches, we can skip saving the FPU state.

As explained above, waking a sleeping thread on a different CPU requires an IPI. These are expensive, and even more so on virtual machines, where both sending and receiving interrupts cause exits to the hypervisor. As an optimization, idle CPUs spend some time before halting in polling state, where they ask not to be sent these IPIs, and instead poll the wakeup bitmask. This optimization can almost eliminate the expensive IPIs in the case where two threads on two CPUs wait for one another in lockstep.

3 Beyond the Linux APIs

In this section, we explore what kind of new APIs a single-application OS like OS^v might have beyond the standard Linux APIs, and discuss several such extensions which we have already implemented as well as their benefits.

The biggest obstacle to introducing new APIs is the need to modify existing applications or write new applications. One good way around this problem is to focus on efficiently running a *runtime environment*, such as the Java Virtual Machine (JVM), on OS^v. If we optimize the JVM itself, any application run inside this JVM will benefit from this optimization.

As explained in the previous section, OS^v can run unmodified Linux programs, which use the Linux APIs — a superset of the POSIX APIs. We have lowered the overhead of these APIs, as described in the previous section and quantified in the next section. One of the assumptions we have made is that OS^v runs a single application, in a single address space. This allowed us to run “system calls” as ordinary functions, reducing their overhead.

However, in this section we show that there remain significant overheads and limitations inherent in the Linux APIs, which were designed with a multi-process multi-user operating system in mind. We propose to reduce these remaining overheads by designing new APIs specifically for applications running on a single-process OS like OS^v.

The socket API, in particular, is rife with such overheads. For example, a socket read or write necessarily copies the data, because on Linux the kernel cannot share packet buffers with user space. But on a single-address-space OS, a new zero-copy API can be devised where the kernel and user space share the buffers. For packet-processing applications, we can adopt a netmap-like API [23]. The OS^v kernel may even expose the host’s virtio rings to the application (which is safe when we have a single application), completely eliminating one layer of abstraction. In Section 4 we demonstrate a Memcached implementation which uses a non-POSIX packet processing API to achieve a 4-fold increase of throughput compared to the traditional Memcached using the POSIX socket APIs.

Another new API benefiting from the single-application nature of OS^v is one giving the application direct access to the page table. Java’s GC performance, in particular, could benefit: The Hotspot JVM uses a data structure called a *card table* [22] to track write accesses to references to objects. To update this card table to mark memory containing that reference as dirty, the code generated by the JVM has to be followed by a “write barrier”. This additional code causes both extra instructions and cache line bounces. However, the MMU already tracks write access to memory. By giving the JVM access to the MMU, we can track reference modifications without a separate card table or write barriers. A similar strategy is already employed by Azul C4 [27], but it requires heavy modifications to the Linux memory management system.

In the rest of this section, we present two new non-Linux features which we implemented in OS^v. The first feature is a *shrinker* API, which allows the application and the kernel to share the entire available memory. The second feature, the *JVM balloon*, applies the shrinker idea to an unmodified Java Virtual Machine, so that instead of manually chosing a heap size for the JVM, the heap is automatically resized to fill all memory which the kernel does not need.

3.1 Shrinker

The shrinker API allows the application or an OS component to register callback functions that OS^v will call when the system is low on memory. The callback function is then responsible for freeing some of that application or component’s memory. Under most operating systems, applications or components that maintain a dynamic cache, such as a Memcached cache or VFS page cache, must statically limit its size to a pre-defined amount of memory or number of cache entries. This imposes sometimes contradicting challenges: not to consume more memory than available in the system and not to strangle other parts of the system, while still taking advantage of the available memory. This gets even more challenging when there are a few heavy memory consumers in the system that work in a bursty manner wherein the memory needs to “flow” from one application or component to another depending on demand. The shrinker API provides an adaptable solution by allowing applications and components to handle memory pressure as it arises, instead of requiring admininstrators to tune in advance.

We have demonstrated the usefulness of the shrinker API in two cases — Memcached [6], and the JVM. Ordinarily, Memcached requires the in-memory cache size to be specified (with the "-m" option) and the JVM requires the maximum heap size to be specified (the "-Xmx" option). Setting these sizes manually usually results in wasted VM memory, as the user decreases the cache or heap size to leave "enough" memory to the OS. Our Memcached re-implementation described in Section 4 uses the shrinker API and does not need the "-m" option: it uses for its cache all the memory which OS^{v} doesn't need. We can similarly modify the JVM to use the shrinker to automatically size its heap, and even achieve the same on an unmodified JVM, as we will explain now.

3.2 JVM Balloon

The *JVM balloon* is a mechanism we developed to automatically determine the JVM heap size made available to the application. Ballooning is a widely used mechanism in hypervisors [30, 31] and the JVM balloon draws from the same core idea: providing efficient dynamic memory placement and reducing the need to do complex planning in advance. OS^{v}'s JVM balloon is designed to work with an unmodified JVM. As a guest-side solution, it will also work on all supported hypervisors.

It is possible to modify the JVM code to simplify this process. But the decision to run it from the OS side allows for enhanced flexibility, since it avoids the need to modify the various extant versions and vendor-implementations of the JVM.

The JVM allocates most of its memory from its heap. This area can grow from its minimum size but is bounded by a maximum size, both of which can be specified by initialization parameters. The size of the JVM heap directly influences performance for applications since having more memory available reduces occurrences of GC cycles.

However, a heap size that is too big can also hurt the application since the OS will be left without memory to conduct its tasks — like buffering a large file — when it needs to. Although any modern OS is capable of paging through the virtual-memory system, the OS usually lacks information during this process to make the best placement decision. A normal OS will see all heap areas as pages whose contents cannot be semantically interpreted. Consequently, it is forced to evict such pages to disk, which generates considerable disk activity and suboptimal cache growth. At this point an OS that is blind to the semantic content of the pages will usually avoid evicting too much since it cannot guarantee that those pages will not be used in the future. This results in less memory being devoted to the page cache, where it would potentially bring the most benefit. We quantify this effect in Section 4, and show that OS^{v}'s JVM balloon allows pages to be discarded without any disk activity.

OS^{v}'s approach is to allocate almost all available memory to the JVM when it is started [1], therefore setting that memory as the *de facto* JVM maximum heap. The OS allocations can proceed normally until pressure criteria are met.

Upon pressure, OS^{v} will use JNI [13] to create an object in the JVM heap with a size big enough to alleviate that pressure and acquire a reference to it. The object chosen is a `ByteArray`, since these are laid down contiguously in memory and it is possible to acquire a pointer to their address from JNI.

This object is referenced from the JNI, so a GC will not free it and at this point the heap size is effectively reduced by the size of the object, forcing the JVM to count on a smaller heap for future allocations. Because the balloon object still holds the actual pages as backing storage, the last step of the ballooning process is to give the pages back to the OS by unmapping that area. The JVM cannot guarantee or force any kind of alignment for the object, which means that in this process some memory will be wasted: it will neither be used the Java application nor given back to the OS. To mitigate this we use reasonably large minimum balloon sizes (128MB).

Balloon movement

The reference to the object, held by OS^{v}, guarantees that the object will not be disposed by the JVM or taken into account when making collection decisions. However, it does not guarantee that the object is never touched again. When the JVM undergoes a GC cycle, it moves the old objects to new locations to open up space for new objects to come. At this point, OS^{v} encounters a page fault.

OS^{v} assumes that nothing in the JVM directly uses that object, and therefore is able to make the following assumptions about page faults that hit the balloon object:

- all reads from it are part of a copy to a new location,
- the source and destination addresses correspond to the same offset within the object,
- whenever that region is written to, it no longer holds the balloon.

With that in mind, OS^{v}'s page fault handler can decode the copy instruction — usually a `rep mov` in x86 — and find its destination operand. It then recreates the balloon in the destination location and updates all register values

[1] 90% in the current implementation

to make the copier believe the copy was successfully conducted. OS^v's balloon mechanism is expected to work with any JVM or collector in which these assumptions hold.

The old location is kept unmapped until it is written to. This has both the goal of allowing the remap to be lazy, and to correctly support GCs that may speculatively copy the object to more than one location. Such is the case, for instance, for OpenJDK's Parallel Scavenge Garbage Collector.

4 Evaluation

We conducted some experiments to measure the performance of OSv as a guest operating system, and demonstrate improvement over a traditional OS: Linux. In all runs below, for "Linux" we used a default installation of Fedora 20 with the `iptables` firewall rules cleared. We look at both micro-benchmarks measuring the performance of one particular feature, and macro-benchmarks measuring the overall performance of an application.

The host used in the benchmarks was a 4-CPU 3.4GHz Intel® Core™ i7-4770 CPU, 16GB of RAM, with an SSD disk. The host was running Fedora 20 Linux and the KVM hypervisor.

Macro Benchmarks

Memcached is a high-performance in-memory key-value storage server [6]. It is used by many high-profile Web sites to cache results of database queries and prepared page sections, to significantly boost these sites' performance. We used the *Memaslap* benchmark to load the server and measure its performance. Memaslap runs on a remote machine (connected to the tested host with a direct 40 GbE cable), sends random requests (concurrency 120), 90% `get` and 10% `set`, to the server and measures the request completion rate. In this test, we measured a single-vCPU guest running Memcached with one service thread. Memcached supports both UDP and TCP protocols — we tested the UDP protocol which is considered to have lower latency and overhead [20]. We set the combination of Memcached's cache size (5 GB) and memaslap test length (30 seconds) to ensure that the cache does not fill up during the test.

Table 1 presents the results of the *memaslap* benchmark, comparing the same unmodified Memcached program running on OS^v and Linux guests. We can see that Memcached running on OS^v achieves 22% higher throughput than when running on Linux.

One of the stated goals of OS^v was that an OS^v guest boots quickly, and has a small image size. Indeed, we measured the time to boot OS^v and Memcached, until Memcached starts serving requests, to be just 0.6 seconds. The guest image size was just 11MB. We believe that both numbers can be optimized further, e.g., by using ramfs instead of ZFS (Memcached does not need persistent storage).

Guest OS	**Transactions / sec**	**Score**
Linux	104394	1
OS^v	127275	1.22

Table 1: Memcached and Memaslap benchmark

In Section 3 we proposed to further improve performance by implementing in OS^v new networking APIs with lower overheads than the POSIX socket APIs. To test this direction, we re-implemented part of the Memcached protocol (the parts that the memaslap benchmark uses). We used a packet-filtering API to grab incoming UDP frames, process them, and send responses in-place from the packet-filter callback. As before, we ran this application code in a single-vCPU guest running OS^v and measured it with memaslap. The result was 406750 transactions/sec — 3.9 times the throughput of the baseline Memcached server on Linux.

SPECjvm2008 is a Java benchmark suite containing a variety of real-life applications and benchmarks. It focuses on the performance of the JVM executing a single application, and reflects the performance of CPU- and memory-intensive workloads, having low dependence on file I/O and including no network I/O across machines.

SPECjvm2008 is not only a performance benchmark, it is also a good correctness test for OS^v. The benchmarks in the suite use numerous OS features, and each benchmark validates the correctness of its computation.

Table 2 shows the scores for both OS^v and Linux for the SPECjvm2008 benchmarks. For both guest OSs, the guest is given 2GB of memory and two vCPUs, and the benchmark is configured to use two threads. The JVM's heap size is set to 1400MB.

Benchmark	OS^v	Linux	Benchmark	OS^v	Linux
Weighted average	**1.046**	**1.041**	sor.large	27.3	27.1
compiler.compiler	377	393	sparse.large	27.7	27.2
compiler.sunflow	140	149	fft.small	138	114
compress	111	109	lu.small	216	249
crypto.aes	57	56	sor.small	122	121
crypto.rsa	289	279	sparse.small	159	163
crypto.signverify	280	275	monte-carlo	159	150
derby	176	181	serial	107	107
mpegaudio	100	100	sunflow	56.6	55.4
fft.large	35.5	32.8	xml.transform	251	247
lu.large	12.2	12.2	xml.validation	480	485

Table 2: SPECjvm2008 — higher is better

We did not expect a big improvement, considering that SPECjvm2008 is computation-dominated with relatively little use of OS services. Indeed, on average, the SPECjvm2008 benchmarks did only 0.5% better on OS^v

than on Linux. This is a small but statistically-significant improvement (the standard deviation of the weighted average was only 0.2%). OS^v did slightly worse than Linux on some benchmarks (notably those relying on the filesystem) and slightly better on others. We believe that with further optimizations to OS^v we can continue to improve its score, especially on the lagging benchmarks, but the difference will always remain small in these computation-dominated benchmarks.

Micro Benchmarks

Network performance: We measured the network stack's performance using the Netperf benchmark [11] running on the host. Tables 3 and 4 shows the results for TCP and UDP tests respectively. We can see that OS^v consistently outperforms Linux in the tests. RR (request/response) is significantly better for both TCP and UDP, translating to 37%-47% reduction in latency. TCP STREAM (single-stream throughput) is 24%-25% higher for OS^v.

Test	STREAM (Mbps)	RR (Tps)
Linux UP	44546 ± 941	45976 ± 299
Linux SMP	40149 ± 1044	45092 ± 1101
OS^v UP	55466 ± 553	74862 ± 405
OS^v SMP	49611 ± 1442	72461 ± 572

Table 3: Netperf TCP benchmarks: higher is better

Test	RR (Tps)
Linux UP	44173 ± 345
Linux SMP	47170 ± 2160
OS^v UP	82701 ± 799
OS^v SMP	74367 ± 1246

Table 4: Netperf UDP benchmarks: higher is better

JVM balloon: To isolate the effects of the JVM balloon technique described in Section 3.2, we wrote a simple microbenchmark in Java to be run on both Linux and OS^v. It consists of the following steps:

1. Allocate 3.5 GB of memory in 2MB increments and store them in a list,
2. Remove from the list and write each 2MB buffer to a file sequentially until all buffers are exhausted,
3. Finally read that file back to memory.

In both guest OSs, the application ran alone in a VM with 4GB of RAM. For OS^v, the JVM heap size was automatically calculated by the balloon mechanism to 3.6 GB. For Linux, the same value was manually set.

As shown in Table 5, OS^v fared better in this test than Linux by around 35%. After the first round of allocations the guest memory is almost depleted. As Linux needs more memory to back the file it has no option but to evict JVM heap pages. That generates considerable disk activity, that not only is detrimental per se, but will in this particular moment compete with the application disk writes.

We observe that not only is the execution slower on Linux, it also has a much higher standard deviation. This is consistent with our expectation. Aside from deviations arising from the I/O operations themselves, the Linux VM lacks information to make the right decision about which pages is best to evict.

Guest OS	Total (sec)	File Write (sec)	File Read (sec)
Linux	40 ± 6	27 ± 6	10.5 ± 0.2
OS^v	26 ± 1	16 ± 1	7.4 ± 0.2

Table 5: JVM balloon micro-benchmark: lower is better

OS^v can be more agressive when discarding pages because it doesn't have to evict pages to make room for the page cache, while Linux will be a lot more conservative in order to avoid swap I/O. That also speeds up step 3 (”File Read”), as can be seen in Table 5. In the absence of eviction patterns, both Linux and OS^v achieve consistent results with a low deviation. However, Linux reaches this phase with a smaller page cache to avoid generating excessive disk activity. OS^v does not need to make such compromise, leading to a 30% performance improvement in that phase alone.

Context switches: We wrote a context-switch microbenchmark to quantify the claims made earlier that thread switching is significantly cheaper on OS^v than it is on Linux. The benchmark has two threads, which alternate waking each other with a pthreads condition variable. We then measure the average amount of time that each such wake iteration took.

The benchmark is further subdivided into two cases: In the “colocated” case, the two alternating threads are colocated on the same processor, simulating the classic uniprocessor context switch. In the “apart” case, the two threads are pinned to different processors.

Guest OS	Colocated	Apart
Linux	905 ns	13148 ns
OS^v	328 ns	1402 ns

Table 6: Context switch benchmark

The results are presented in Table 6. It shows that thread switching is indeed much faster in OSv than in Linux — between 3 and 10 times faster. The “apart” case is especially helped in OSv by the last optimization described in 2.4, of idle-time polling.

5 Related Work

Containers [26, 3] use a completely different approach to eliminate the feature duplication of the hypervisor and guest OS. They abandon the idea of a hypervisor, and instead provide *OS-level virtualization* — modifying the host OS to support isolated execution environments for applications while sharing the same kernel. This approach improves resource sharing between guests and lowers per-guest overhead. Nevertheless, the majority of IaaS clouds today use hypervisors. These offer tenants better-understood isolation and security guarantees, and the freedom to choose their own kernel.

Picoprocesses [4] are another contender to replace the hypervisor. While a containers' host exposes to its guests the entire host kernel's ABI, picoprocesses offer only a bare-minimum API providing basic features like allocating memory, creating a thread and sending a packet. On top of this minimal API, a library OS is used to allow running executables written for Linux [9] or Windows [21]. These library OSs are similar to OS$^{\text{v}}$ in that they take a minimal host/guest interface and use it to implement a full traditional-OS ABI for a single application, but the implementation is completely different. For example, the picoprocess POSIX layer uses the host's threads, while OS$^{\text{v}}$ needs to implement threads and schedule these threads on its own.

If we return our attention to hypervisors, one known approach to reducing the overheads of the guest OS is to take an existing operating system, such as a Linux distribution, and trim it down as much as possible. Two examples of this approach are CoreOS and Tiny Core Linux. OS$^{\text{v}}$ differs from these OSs in that it is a newly designed OS, not a derivative of Linux. This allowed OS$^{\text{v}}$ to make different design decisions than Linux made, e.g., our choice not to use spinlocks, or to have a single address space despite having an MMU.

While OS$^{\text{v}}$ can run applications written in almost any language (both compiled and high-level), some VM OSs focus on running only a single high-level language. For example, *Erlang on Xen* runs an Erlang VM directly on the Xen hypervisor. *Mirage OS* [14] is a library OS written in OCaml that runs on the Xen hypervisor. It takes the idea of a library OS to the extreme where an application links against separate OS service libraries and unused services are eliminated from the final image by the compiler. For example, a DNS server VM image can be as small as 200 KB.

Libra [1] is a library OS for running IBM's J9 JVM in a VM. Libra makes the case that as JVM already has sandboxing, a memory model, and a threading model, a general purpose OS is redundant. However, Libra does not replace the whole OS but instead relies on Linux running in a separate hypervisor partition to provide networking and filesystem.

ClickOS [15] is an optimized operating system for VMs specializing in network processing applications such as routing, and achieves impressive raw packet-per-second figures. However, unlike OS$^{\text{v}}$ which runs on multiple hypervisors, ClickOS can only run on Xen, and requires extensive modifications to Xen itself. Additionally, ClickOS is missing important functionality that OS$^{\text{v}}$ has, such as support for SMP guests and a TCP stack.

6 Conclusions and Future Work

We have shown that OS$^{\text{v}}$ is, in many respects, a more suitable operating system for virtual machines in the cloud than are traditional operating systems such as Linux. OS$^{\text{v}}$ outperforms Linux in many benchmarks, it makes for small images, and its boot time is barely noticeable. OS$^{\text{v}}$ is a young project, and we believe that with continued work we can further improve its performance.

While OS$^{\text{v}}$ improves the performance of existing applications, some of the most dramatic improvements we've seen came from adding non-POSIX API to OS$^{\text{v}}$. For example, the shrinker API allows an OS$^{\text{v}}$-aware application to make better use of available memory, and a packet-filtering APIs reduces the overheads of the standard socket APIs. We plan to continue to explore new interfaces to add to OS$^{\text{v}}$ to further improve application performance. Areas of exploration will include network APIs and cooperative scheduling.

Instead of modifying many individual applications, a promising future direction is to modify a runtime environment, such as the JVM, on which many applications run. This will allow us to run unmodified applications, while still benefiting from new OS$^{\text{v}}$ APIs. The JVM balloon we presented is an example of this direction.

Finally, we hope that the availability of OS$^{\text{v}}$, with its small modern code and broad usability (not limited to specific languages, hypervisors or applications) will encourage more research on operating systems for VMs.

7 Acknowledgments

Being an Open Source project, we would like to thank our community contributors, especially those ones working as volunteers.

8 Availability

OS$^{\text{v}}$ is BSD-licensed open source, available at:

`https://github.com/cloudius-systems/osv`

More information is available at `http://osv.io/`.

References

[1] AMMONS, G., APPAVOO, J., BUTRICO, M., DA SILVA, D., GROVE, D., KAWACHIYA, K., KRIEGER, O., ROSENBURG, B., VAN HENSBERGEN, E., AND WISNIEWSKI, R. W. Libra: a library operating system for a JVM in a virtualized execution environment. In *Proceedings of the 3rd international conference on Virtual execution environments* (2007), ACM, pp. 44–54.

[2] ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (1990), 6–16.

[3] DES LIGNERIS, B. Virtualization of Linux based computers: the Linux-VServer project. In *International Symposium on High Performance Computing Systems and Applications* (2005), IEEE, pp. 340–346.

[4] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008), pp. 339–354.

[5] ENGLER, D. R., KAASHOEK, M. F., ET AL. Exokernel: An operating system architecture for application-level resource management. In *ACM SIGOPS Operating Systems Review* (1995), vol. 29, pp. 251–266.

[6] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal*, 124 (2004).

[7] FRIEBEL, T., AND BIEMUELLER, S. How to deal with lock holder preemption. *Xen Summit North America* (2008).

[8] GIDENSTAM, A., AND PAPATRIANTAFILOU, M. LFthreads: A lock-free thread library. In *Principles of Distributed Systems*. Springer, 2007, pp. 217–231.

[9] HOWELL, J., PARNO, B., AND DOUCEUR, J. R. How to run POSIX apps in a minimal picoprocess. In *2013 USENIX ATC* (2013), pp. 321–332.

[10] JACOBSON, V., AND FELDERMAN, R. Speeding up networking. *Linux Conference Australia* (2006).

[11] JONES, R. A. A network performance benchmark (revision 2.0). Tech. rep., Hewlett Packard, 1995.

[12] KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Summer* (1986), vol. 86, pp. 238–247.

[13] LIANG, S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[14] MADHAVAPEDDY, A., MORTIER, R., ROTSOS, C., SCOTT, D., SINGH, B., GAZAGNAIRE, T., SMITH, S., HAND, S., AND CROWCROFT, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013), ACM.

[15] MARTINS, J., AHMED, M., RAICIU, C., OLTEANU, V., HONDA, M., BIFULCO, R., AND HUICI, F. ClickOS and the art of network function virtualization. In *USENIX NSDI* (2011).

[16] MASSALIN, H., AND PU, C. A lock-free multiprocessor OS kernel. *ACM SIGOPS Operating Systems Review 26*, 2 (1992), 108.

[17] MCKENNEY, P. E., AND SLINGWINE, J. D. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems* (1998), pp. 509–518.

[18] MEGIDDO, N., AND MODHA, D. Outperforming LRU with an adaptive replacement cache algorithm. *Computer 37*, 4 (April 2004), 58–65.

[19] MICHAEL, M. M., AND SCOTT, M. L. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 1–26.

[20] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., ET AL. Scaling Memcache at Facebook. In *USENIX NSDI* (2013).

[21] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. *ACM SIGPLAN Notices 46*, 3 (2011), 291–304.

[22] PRINTEZIS, T. Garbage collection in the Java HotSpot virtual machine, 2005.

[23] RIZZO, L. netmap: a novel framework for fast packet I/O. In *USENIX ATC* (2012).

[24] ROMER, T. H., OHLRICH, W. H., KARLIN, A. R., AND BERSHAD, B. N. Reducing TLB and memory overhead using online superpage promotion. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on* (1995), IEEE, pp. 176–187.

[25] RUSSELL, R. virtio: towards a de-facto standard for virtual I/O devices. 95–103.

[26] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 275–287.

[27] TENE, G., IYENGAR, B., AND WOLF, M. C4: the continuously concurrent compacting collector. *ACM SIGPLAN Notices 46*, 11 (2011), 79–88.

[28] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *Virtual Machine Research and Technology Symposium* (2004), pp. 43–56.

[29] VMWARE INC. ESX Server 2 - architecture and performance implications. Tech. rep., VMWare, 2005.

[30] WALDSPURGER, C. A. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev. 36*, SI (Dec. 2002), 181–194.

[31] ZHAO, W., WANG, Z., AND LUO, Y. Dynamic memory balancing for virtual machines. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 37–47.

Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications

Xiaoning Ding[*] Phillip B. Gibbons[†] Michael A. Kozuch[†] Jianchen Shan[*]
[*]*New Jersey Institute of Technology* [†]*Intel Labs Pittsburgh*

Abstract

As the number of cores in a multicore node increases in accordance with Moore's law, the question arises as to what are the costs of virtualized environments when scaling applications to take advantage of larger core counts. While a widely-known cost due to preempted spinlock holders has been extensively studied, this paper studies another cost, which has received little attention. The cost is caused by the intervention from the VMM during synchronization-induced idling in the application, guest OS, or supporting libraries—we call this the *blocked-waiter wakeup (BWW)* problem.

The paper systematically analyzes the cause of the BWW problem and studies its performance issues, including increased execution times, reduced system throughput, and performance unpredictability. To deal with these issues, the paper proposes a solution, *Gleaner*, which integrates idling operations and imbalanced scheduling as a mitigation to this problem. We show how *Gleaner* can be implemented without intrusive modification to the guest OS. Extensive experiments show that *Gleaner* can effectively reduce the virtualization cost incurred by blocking synchronization and improve the performance of individual applications by 16x and system throughput by 3x.

1 Introduction

Virtualized environments are ubiquitous, and are increasingly run on multicore nodes, particularly in the cloud. Amazon EC2's *CC2* and *CR1* instances, for example, offer 32 virtual CPUs (vCPUs) running on two 8-core Intel® Xeon® E5-2670 processors with hyperthreading [2]. When computational workloads execute in these virtualized environments, each "guest" operating system (OS) is presented with a VM instance comprised of a set of vCPUs on which it schedules application threads. The virtual machine manager (VMM), or hypervisor, independently schedules the virtual CPUs onto the physical CPUs (pCPUs) of the host machine. The VMM is often required to time-share the pCPUs among co-running VMs, and may deschedule a vCPU belonging to one VM in favor of a vCPU belonging to another VM. Unfortunately, the resulting behavior of the vCPU abstraction does not always match the behavior of physical CPUs for which the applications and OS are designed. In particular, applications and OS may expect that busy CPUs can make continuous progress in parallel and that idle CPUs are ready for use immediately.

The mismatch between the vCPU abstraction and pCPU behavior introduces great challenges to synchronization and causes serious performance issues, particularly for multithreaded applications running on multicore VMs. One such issue, which has been extensively studied [6, 24, 26], is known as the Lock-Holder Preemption problem (LHP). LHP surfaces when, for example, a vCPU is descheduled from the host platform while the thread currently executing on that vCPU is holding a lock. Other threads running in that VM that are waiting on the lock may be prevented from making progress until the descheduled vCPU is rescheduled, even though their vCPU resources are active. Several software [6, 24, 26] and hardware (such as the pause-loop-exiting (PLE) support on Intel® processors [21]) solutions have been proposed to mitigate this issue.

As the number of cores per socket continues to increase in accordance with Moore's law, though, a natural question arises: *Does the mismatch between the vCPU abstraction and real hardware CPUs impose additional "hidden" performance issues associated with synchronization that may prevent multithreaded applications from taking full advantage of larger core counts?*

The Blocked-Waiter Wakeup Problem. One such issue can be viewed somewhat as the dual of the LHP problem; we call this issue the *Blocked-Waiter Wakeup (BWW)* problem. The BWW problem may arise anytime that a multithreaded application using blocking synchronization executes in a virtualized environment, and it can cause increased execution times, reduced system throughput, and performance unpredictability.

In non-virtualized systems, when an application's thread blocks waiting for a resource to be freed, the CPU resource occupied by that thread is typically returned to the guest OS; if the OS has no other work to schedule onto the CPU, it may halt that CPU, allowing it to enter a low-power state. Later, when the resource is freed, the OS wakes the halted CPU by a signal from another CPU, for example, by issuing an Inter-Processor Interrupt (IPI). Unfortunately, when this occurs in a virtualized environment, the idle vCPU is not simply in a halt state, waiting to be awoken at a moment's notice. Instead, it has often been de-scheduled, and the wakeup IPI, which is comparatively lightweight on physical hardware, must now cause a VM trap, invoke the VMM scheduler, and cause the blocked vCPU to be rescheduled. Furthermore, unlike an idle physical core, which is ready for immediate use once awoken, an idle vCPU can be delayed waiting for a pCPU to free up.

As we show, the increased latency associated with this wakeup path can significantly increase the execution time of virtualized multithreaded applications (e.g., by 517% for dedup on 16 pCPUs) relative to their unvirtualized performance, even when the application's VM has dedicated use of the underlying physical hardware and does not use emulated resources. Note that the offending synchronization may not even be explicit at the application level—in our experiments, we found that the problematic blocking synchronization may arise in the guest OS code.

Our Solution: Gleaner. To mitigate the detrimental performance effects of the BWW problem, we propose an approach, called *Gleaner*, which consolidates short idle periods on multiple vCPUs into long idle periods on fewer cores, thereby lessening the frequency that vCPUs enter/exit idle loops. Two key insights motivate this as a solution. First, applications vulnerable to the BWW problem are likely to see many such idle-busy cycles and, hence, may be under-utilizing their CPU resources. Second, activating/switching threads at the OS level within the VM is much lower overhead than activating/switching vCPUs at the VMM level outside the VM. Our experiments with a prototype *Gleaner* implementation indicate that this approach significantly mitigates the BWW problem.

To date, the BWW problem has been under-studied. It was first discussed briefly as part of a broader technical report by Song, *et al.* [22] and then expanded by our previous short work [4]. This paper represents the first full work dedicated to the problem and makes the following contributions: (1) a systematic characterization of the BWW problem, an important issue for virtualized multicore systems, (2) the design of an effective approach, *Gleaner*, for mitigating the BWW problem, and (3) an experimental validation of *Gleaner*'s effectiveness by demonstrating improvements of application performance by up to 16x and improvements of system throughput by up to 3x.

Table 1: Performance of dedup under *Native* (unvirtualized), *Dedicated* (virtualized, no other load), and *Shared* (two equal sized VMs) configurations.

			Slowdown	
		run	Relative	Relative to
	cores	time (s)	to Native	Dedicated
Native	1	23.5	–	–
	4	7.1	–	–
	16	7.6	–	–
Dedicated	1	26.4	1.1	–
	4	13.5	1.9	–
	16	46.9	6.2	–
Shared w/ streamcluster	4	52.3	7.4	3.9
	16	81.6	10.7	1.7
Shared w/ matmul	4	65.1	9.2	4.8
	16	577.2	75.9	12.3

2 Motivating Example

Table 1 presents an illustrative example of the above virtualization problems with the dedup benchmark from the PARSEC-3.0 suite. We measured the performance[1] of dedup under three settings. In the *Native* setting, dedup executed alone on physical hardware. In the *Dedicated* hardware setting, dedup ran in a VM with a dedicated pCPU allocated to each vCPU. In the *Shared* hardware setting, dedup executed in a VM sharing hardware resources with another VM (in which either streamcluster or matmul ran[2]), both sized to occupy the entire host.

The table shows that the virtualization penalty increases as the number of vCPUs scales up– reaching a factor of 6.2 for 16 cores (*Dedicated* scenario). We observe similar trends for other PARSEC benchmarks, though the penalties are not as dramatic (Section 7.1).

The performance of the *Shared* setting indicates that dedup suffers even more due to hardware resource contention than one might expect. In the experiment, each of the two VMs has the same number of vCPUs as the number of pCPUs and contend for all the hardware CPU resources. With two VMs of the same size competing for the same set of resources, one may expect the slowdown to be approximately 2X relative to the correspond-

[1] Full details of the experimental setup used throughout this paper appear in Section 7.

[2] streamcluster is another benchmark in PARSEC-3.0. matmul is a micro-benchmark multiplying two matrices of 8000×8000 integers.

ing *Dedicated* setting by assuming that each VM would get half of the physical CPU time. However, as shown in the table, the slowdown factor is typically more than 2X and can reach as high as 12.3X relative to *Dedicated*. The table also shows that the degree to which dedup slows down can be greatly affected by the application running in the other VM. Note also that the 16-vCPU execution times differ for the two co-running applications significantly more than they do in the 4-vCPU experiments. We explore these effects more deeply in Section 7.2.

3 Analysis of the BWW Problem

We investigated the possible causes for the performance degradation and variation, and discovered that the applications suffering the most were the ones in which vCPUs were frequently idling due to blocking synchronization in either the benchmarks or the guest system software (OS or supporting libraries).

There are two basic types of inter-thread synchronization primitives: *spinning*, where a waiting thread repeatedly checks some condition to determine if it can continue (possibly remaining in user space), and *blocking*, where a waiting thread yields its execution resources and relies on system software to wake it up when it can continue executing. Often, synchronization libraries combine the two approaches: a thread spins for a brief period of time, and if the desired condition has not been satisfied, the thread blocks.

One effect of blocking synchronization is that the number of *active* application threads may change dynamically, and consequently, the number of cores actively employed by that application may change accordingly. When the number of active threads drops below the number of active cores, some cores will become idle. When the number of active threads increases beyond the number of active cores, idle cores must be activated. For example, when a thread calls *pthread_mutex_lock()* to request a lock that is held by another thread, it will block itself through appropriate library/system calls, waiting for the release of the lock. If there are no other threads ready to run in the system, the core running the thread becomes idle. With conventional OS design, an idle core executes the idle loop, which typically calls a special instruction (e.g., `HLT` on Intel® 64 and IA-32 architecture (“x86”) platforms) that may lead to the core entering a low power state. When the lock is released, the threads waiting for it are woken up. To maximize throughput, the OS may activate idle cores to schedule waking threads onto them.

In a virtualized environment, some of the operations executed by guest software during blocking synchronization routines must be handled by the VMM, even though they would be carried out directly by hardware in a non-virtualized environment. When software issues the special instruction to place a particular core in the idle state, that processor will raise an exception and trap into the VMM. The VMM may take this opportunity to reschedule other vCPUs—perhaps from other VMs—onto the idling physical core. Thus, the “low power” mode for a vCPU is actually the suspension of its execution. When a thread is again ready to run on that vCPU, the VMM must activate the vCPU by rescheduling it onto a physical core. This suffers much higher cost than it does in a non-virtualized environment, in which switching a core back from low power mode can be very fast. For example, switching from C1 to C0 states takes less than 1 µs on contemporary Intel® Xeon® CPUs.[3]

Table 2: Time to wake up a thread on a physical machine (PM) and a virtual machine (VM) under different settings. The last column shows in parentheses the major operations needed to wake up a thread in a VM: RT=rescheduling thread, IPI=handling reschedule IPI, RV=rescheduling vCPU, and PV=preempting a vCPU.

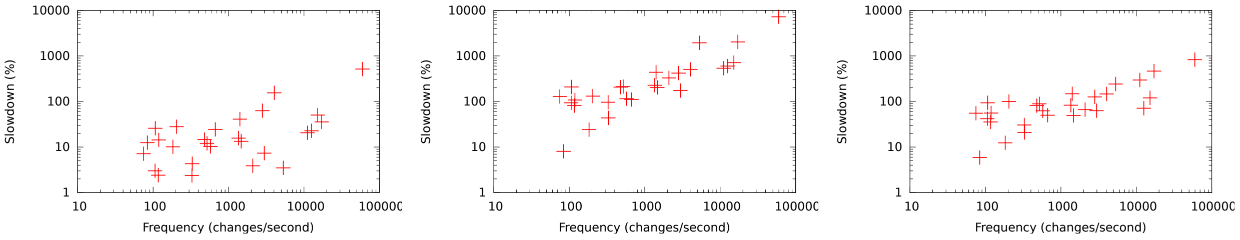
Setting	PM	VM
A: same core	4 µs	6 µs (RT)
B: diff cores, spinning	8 µs	17 µs (IPI, RT)
C: diff cores, blocking	8 µs	37 µs (IPI, RV, RT)
D: diff cores, blocking (2 apps or 2 VMs)	17 µs	>96 µs (IPI, PV, RV, RT)

This is the heart of the Blocked-Waiter Wakeup (BWW) problem: in a virtualized environment, the increased cost of thread wakeup operations may significantly degrade overall performance. To see this, we first show the increased cost of wakeup operations (Section 3.1) and then the correlation between performance and the frequency of idleness transitions (Section 3.2).

3.1 Blocking Synchronization Cost in VMs

To understand the cost of blocking synchronization in VMs and analyze how the cost is increased by switching and scheduling vCPUs, we report the time to wake up a thread blocked in *pthread_mutex_lock()* on both a physical machine (PM) and a virtual machine (VM) under four different settings, as shown in Table 2. In setting *A*, the thread calling *pthread_mutex_unlock()* and the thread blocked in *pthread_mutex_lock()* are pinned to the same core on the PM or to the same vCPU in the VM. In the other three settings, the two threads are pinned to different cores on the PM or different vCPUs in the VM. Thus when the thread is blocked in *pthread_mutex_lock()*, the corresponding core/vCPU will become idle. In setting *B*,

[3] Waking up a core from deep sleep modes (e.g., C3 and C4 states) can take more than 100 µs. But these modes are not used unless the system ensures that the core will stay idle for a long time. Waking up a core from C1E state takes about 10 µs, but system administrators often disable the state for better performance [27].



(a) setting *C*

(b) setting *D* with matmul

(c) setting *D* with freqsleep

Figure 1: Correlation between the performance degradation of the SPLASH-2X and PARSEC-3.0 benchmarks when virtualized and the frequency at which vCPUs transition to idle (when run alone).

lower power modes are disabled by keeping idle physical cores polling on the PM and by running low priority threads repeatedly calling *sched_yield()* on the VM. In setting *C*, lower power modes are enabled. Specifically, when a physical core becomes idle it enters C1 state, and when a vCPU becomes idle it calls HLT to suspend itself. In setting *D*, we measure wake-up times when another application is contending for CPU resources. In the PM experiment, we run a matmul thread on every core. In the VM experiment, we run a second VM with a matmul thread on every vCPU; the number of vCPUs in each of the two VMs is same as the number of physical cores.

Table 2 clearly demonstrates that virtualization significantly increases the cost of blocking synchronization. Waking up a thread in a VM has different costs under the four settings due to the different operations involved. Under setting *A*, waking up a thread in the VM involves only a context switch between the threads inside the guest OS. Thus, it incurs similar overhead as on the physical machine. Under setting *B*, waking up a thread on a vCPU is initiated by an IPI (inter-processor interrupt) made by another vCPU in the same VM (e.g., the one observing the *pthread_mutex_unlock()*). In a non-virtualized environment, the IPI is delivered by hardware, but in a virtualized environment, the VMM must intercept and deliver the IPI. Thus, waking up a thread on a VM incurs higher overhead than on a PM. Under setting *C*, waking up the thread in the VM takes 37 µs, **4.6** times the latency for the same operation on the PM. Under setting *D*, waking up the thread in the VM takes the longest time among these settings, at least **2.6** times longer than under setting *C* and **5.6** times longer than either on the PM or waking up a thread on an active vCPU (setting *B*). Waking up a thread in the VM requires a complete switch between vCPUs from different VMs — suspending a vCPU running matmul, activating and rescheduling the vCPU to run the thread calling *pthread_mutex_lock*. A complete vCPU switch in setting *D* incurs a higher cost than resuming a vCPU on an idle physical core in setting *C* not only because it is between different VMs, but also because the working set of a vCPU may be evicted from the pCPU caches and TLBs when it is de-scheduled. The time (96µs) is measured when the vCPU switch takes place immediately after *pthread_mutex_unlock* is called. Depending on vCPU scheduling policies, the vCPU switch may be delayed, further increasing the time to wake up a blocked thread.

3.2 Problems Caused by Accumulated Blocking Synchronization Cost

The runtime overhead incurred by blocking synchronization in a virtualized environment increases with the frequency of active-idle state transitions of application threads, and in particular, state transitions of vCPUs. Such transitions arise frequently for synchronization-intensive applications. The accumulated overhead can significantly degrade performance and increase application performance variation.

To show the correlation between performance degradation and the overhead incurred by blocking synchronization in a virtualized environment, we run SPLASH-2X and PARSEC-3.0 benchmarks under settings *C* and *D*, and measured the frequencies at which vCPUs transition to idle during their executions.[4] Under setting *D*, we run one SPLASH-2X or PARSEC-3.0 benchmark in the first VM, and run either matmul or freqsleep in the second VM. In these experiments, the number of threads in each application, the number of vCPUs in each VM, and the number of physical cores are each 16.

Freqsleep creates a thread on each core, which repeatedly calls *nanosleep(1)*. We select matmul and freqsleep because they have consistent behavior during their execution. So the impact of their execution to the performance of SPLASH-2X and PARSEC-3.0 benchmarks does not change with the execution times of the benchmarks. At the same time, though both matmul and freqsleep can saturate the physical cores, they affect the performance of the benchmarks running in the first VM by different degrees.

As shown in Figure 1, there appears to be a strong correlation between the slowdowns and blocking frequencies. The correlation is more evident when the pCPUs

[4] One may also want to measure the frequencies at which application threads block, but unfortunately, such measurements are challenging because the threads may block inside the OS kernel.

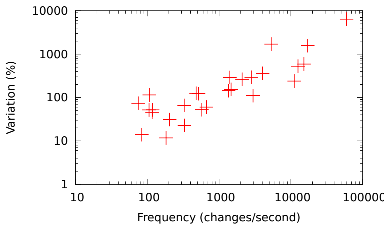


Figure 2: Correlation between the performance variation of the SPLASH-2X and PARSEC-3.0 benchmarks when virtualized and the frequency at which vCPUs transition to idle (when run alone).

are oversubscribed (setting *D*) than when they are not (setting *C*). There are some applications that experience trivial slowdowns under setting *C*, but suffer significant slowdowns under setting *D*. For example, streamcluster is slowed down by 13% under setting *C*. But, it suffers 1660% and 465% slowdowns under setting *D* when matmul and freqsleep run in the second VM, respectively. This can be explained as follows.

To correlate the performance variation of applications in VMs and the frequency at which the vCPUs in the VMs transition their status, for each benchmark, we compare its slowdowns under setting *D* when different applications (i.e., matmul and freqsleep) run in the second VM. Figure 2 shows the absolute value of the difference between the two slowdowns for each benchmark versus the frequency at which the vCPUs become idle during its execution (when run alone). It is evident that benchmarks causing more frequent vCPU status transitions usually experience larger performance variation than other benchmarks.

4 Reducing Harmful Context Switches

To reduce the impact of the BWW problem, we seek to reduce the frequency of harmful switches (those that require the activation of a de-scheduled vCPU). There are at least two possible approaches, *resource retention* and *consolidation scheduling*, and we propose to employ both in an intelligent hybrid design (Section 5).

4.1 Resource Retention

A natural approach to reducing harmful context switches is to avoid releasing resources to lower levels in the software stack. For example, a guest application thread could spin at a problematic synchronization point rather than yield to the guest OS. If the thread becomes unblocked in less time than would be required to transition into the guest OS and back, overall efficiency may be improved by spinning rather than yielding. To avoid application changes, the guest OS may also spin rather than halting. Besides spinning, the guest OS has the additional option of leveraging an operation like the x86 MWAIT instruction, which can place the physical core in a low-power state directly (if permitted by the VMM) such that a simple store to memory will reactivate the core.

However, such resource retention approaches must be employed with some care. Having these operations high in the software stack may lead to under-utilization. In particular, it may prevent layers lower in the stack from improving utilization by reallocating idle resources or placing those resources in a low-power state.

When a system is not oversubscribed and hardware resources are not contended, such idling operations will not hurt overall system performance as the resources were underutilized. However, when a system is oversubscribed, resource retention may prevent other VMs from making better use of those resources and reduce system throughput significantly (by as much as 8x [19]).

To improve utilization, these idling operations can be enhanced with a timeout value such that, if the thread or vCPU cannot change its state back from idling within the timeout period, the occupied resource will be released to the control of lower software layers.[5] However, the fundamental tension between improved individual VM performance and overall system throughput remains, albeit with the timeout value as a potential tuning knob.

4.2 Consolidation Scheduling

The second type of mechanism for reducing harmful context switches is based on the observation that, while context switches due to blocking synchronization may be inevitable, the resolution of such switches need not be the responsibility of lower software layers. If higher levels of software can manage the switches, application performance may be improved without adversely affecting overall throughput.

Schedulers (user-level schedulers or guest OS schedulers) determine how tasks are scheduled on execution entities (threads or vCPUs). They have direct influence on whether and when the execution entities would become idle. Thus, they can be improved to reduce the frequency at which threads or vCPUs transition between busy and idle by coalescing tasks onto fewer resources. When tasks are coalesced, the resource onto which they would be scheduled is more likely to be active.

One aspect of consolidation scheduling can be illustrated with Figures 3(a) and 3(b). In (a), the top schedule shows the guest OS scheduling each of threads 1, 2, and 3 immediately when it is ready to run. Thread 1 becomes ready and is scheduled first. It is blocked before thread 2

[5] In addition to the timeout method, an alternative approach can use heuristics to predict the length of the coming idle period and select the action that incurs lower cost: either spinning/MWAIT or yielding hardware resources to lower layers in the software stack.

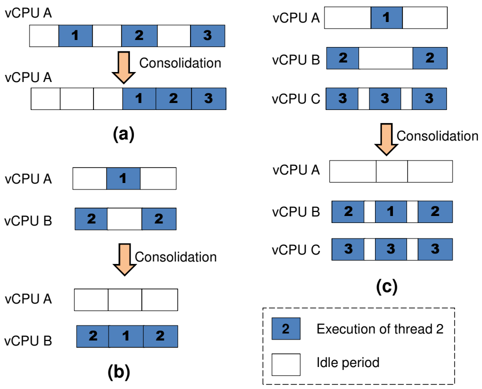


Figure 3: Consolidation scheduling in guest OS (numbers in blocks are thread IDs).

becomes ready, changing the vCPU's state to idle. When thread 2 becomes ready, it is scheduled and the vCPU's state is changed back to busy. The vCPU experiences another round of state changes after thread 2 is blocked and before thread 3 becomes ready. Thus, to reduce state transitions, instead of scheduling a thread immediately when it becomes ready to run, the guest OS scheduler could choose to delay the scheduling of the thread to accumulate enough workloads to keep the vCPU busy for a while, as shown in the bottom schedule. We call this techique to achieve consolidation *delayed scheduling*.

In Figure 3(b), the top schedule shows the guest OS scheduler distributing threads 1 and 2 onto vCPUs A and B for load balance, and neither of the vCPUs can be kept always busy or always idle, increasing the number of state transitions. In contrast, *imbalanced scheduling* consolidates both threads onto one vCPU and keeps the other vCPU idle. While *delayed scheduling* is more suitable to single-vCPU VMs, *imbalanced scheduling* is more suitable to VMs with multiple vCPUs.

While consolidation scheduling tends to improve overall system throughput on an oversubscribed system, it may reduce the resources available to a particular VM and may delay computation and/or overload a busy vCPU if not carefully controlled.

5 Gleaner: Basic Idea and Design

Gleaner is a hybrid approach that combines resource retention and consolidation scheduling techniques to leverage the advantages of both techniques. Resource retention is used to mangae short periods of idleness, and consolidation scheduling is used to both coalesce tasks and reduce the number of long idle periods that resource retention cannot handle efficiently.

The combination can be illustrated with Figure 3(c). Before applying imbalanced scheduling, each vCPU runs one thread. Thread 3 running on vCPU C has short idle periods between its tasks that can be efficiently handled through resource retention without delaying the tasks. Threads 1 and 2 on vCPUs A and B have long idle periods. Holding the hardware with idling operations will incur higher cost than halting the vCPUs. With imbalanced scheduling, the threads are consolidated to vCPU B. Though they cannot fully keep vCPU B busy, running two threads on it reduces the length of idle periods, making resource retention viable. For brevity, the paper refers to the vCPUs with workload threads as *active vCPUs* and refers to other vCPUs as *deactivated vCPUs*.

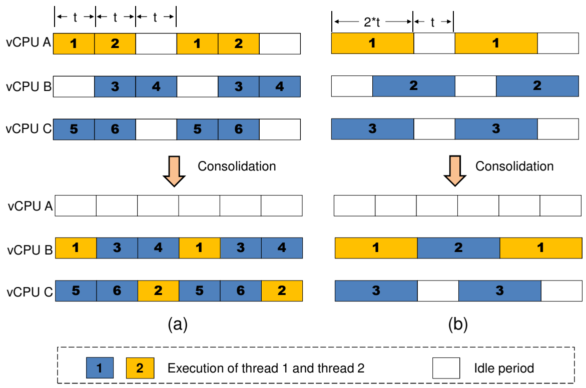


Figure 4: (a) An effective consolidation. (b) Overly-aggressive consolidation overloads vCPU B (the second block of thread 2 remains unscheduled).

To prevent active vCPUs from being overloaded as a result of overly-aggressive consolidation (Figure 4), *Gleaner* monitors the workload in a VM and collects the following time measurements: *computation length*, denoted by l_{comp}, is the length of computation on a vCPU between two consecutive idle periods; *computation granularity*, g_{comp}, is the length of computation in a thread between two consecutive synchronization points; and *length of idle periods*, l_{idle}, is the length of an idle period of a vCPU. Note that *Gleaner* dynamically adjusts the number of active vCPUs in a VM, and measurements need not be collected on deactivated vCPUs. To characterize the workload, *Gleaner* computes averages for these values. In the remainder of the paper, l_{comp}, g_{comp}, and l_{idle} refer to the corresponding average values. For example, for the workload shown on the top of Figure 4(a), l_{comp} is $2t$; both g_{comp} and l_{idle} are t.

Gleaner consolidates workload threads cautiously and gradually. It periodically updates the above measurements, and reduces the active vCPUs one at a time when the following two conditions are satisfied: (1) $l_{comp} \leq \rho \times (N-1) \times l_{idle}$ and (2) $g_{comp} \leq min(\eta \times l_{idle}, min\_time\_slice)$. Here, N is the number of active vCPUs in the VM, and the *load factor*, ρ, and *granularity factor*, η, are tunable values between 0 and 1.

The first condition is to ensure that there is enough idle time on the $N-1$ vCPUs to accommodate the computation on the vCPU to be deactivated. The second condition is to ensure that the computation periods on the to-be-deactivated vCPU are small enough, such that they can be relatively evenly distributed to other active vCPUs and fit into the available idle periods. It also ensures the low overhead of moving threads. *Gleaner* only consolidates workload threads with active computation periods shorter than the minimum timeslice *min_time_slice*, which is selected by the OS to be long enough (e.g., a few milliseconds) to tolerate the overhead of rescheduling threads. For workloads with periods of active computation longer than *min_time_slice*, blocking synchronization usually cannot significantly degrade performance; but cache locality may be an important performance factor. Therefore, instead of consolidating these threads with the imbalanced scheduling technique, *Gleaner* applies resource retention to suppress short idle periods.

Figure 4 uses two examples to explain the necessity to enforce the above conditions. For simplicity, we assume η and ρ are 1 in the examples. In 4(a), the workload running on three vCPUs (as shown on the top) is to be consolidated onto two vCPUs (as shown at the bottom). It meets both conditions. The consolidation does not degrade performance. In 4(b), the workload meets the first condition ($l_{comp} = 2t$ and $l_{idle} = t$), but not the second. Thus, were thread 1 to be migrated to vCPU B, vCPU B would be overloaded, reducing application throughput.

During the consolidation, *Gleaner* keeps monitoring the vCPU utilization of the workload. When it observes a vCPU utilization decrease, which indicates the execution of the workload has slowed down, it stops the consolidation and restores the vCPU that was last deactivated. Then, it adjusts *load factor* and *granularity factor*. Specifically, if the consolidation increases the utilization of a vCPU to 100%, indicating the performance degradation may be caused by the specific vCPU being overloaded (similar to the situation in Figure 4(b)), *granularity factor* is then reduced to $0.9 \times min(\eta, g_{comp}/l_{idle})$; otherwise, *load factor* is reduced to $0.9 \times min(\rho, l_{comp}/((N-1) \times l_{idle}))$. These adjustments are to prevent more consolidation in the future that may degrade performance.

Gleaner maintains the current set of active vCPUs as long as there is not much variation of vCPU utilization and computation granularity. However, a dramatic change in vCPU utilization or computation granularity may indicate a workload change. *Gleaner* must respond to such changes by adjusting the number of active vCPUs to better satisify the resource demand of the workload. Therefore, *Gleaner* activates all the deactivated vCPU in the VM, and distributes the workload on all the vCPUs. Then, it gradually reduces active vCPUs when the above two conditions are met, until it finds a good setting. In our current implementation, a change of vCPU utilization larger than 20%, per-vCPU utilization exceeding 90%, and g_{comp} increased by more than 2x or decreased by more than 50% are each considered as indicators of major change.

6 Gleaner Implementation

A convenient place to implement *Gleaner* is the guest OS. Resource retention techniques can be implemented by modifying the idle driver, and imbalanced scheduling can be achieved by modifying the guest OS scheduler. However, these changes involve intensive modifications to the guest OS. Thus, in this section, we introduce a few techniques to implement *Gleaner* at the user level of the guest OS, to avoid intrusive kernel implementation and enable the adoption into proprietary operating systems.

At user level, idling operations in support of resource retention can be implemented by a *yielding thread* on each active vCPU. A *yielding thread* is a user-level thread that calls the *sched_yield()* system call in a loop. If there are not other threads ready to run on the vCPU, the *sched_yield()* call will return immediately. Otherwise, the *sched_yield()* call relinquishes the vCPU to other threads. Thus, the yielding thread keeps the vCPU active and does not impede the execution of application threads. On systems where the semantics of *sched_yield()* is not fully implemented (e.g., some versions of Linux kernels), the yielding threads should be assigned with the lowest priority possible in the guest OS (e.g., SCHED_IDLE scheduler class in Linux) to avoid hindering workload threads.

While a yielding thread can keep the vCPU active, to support resource retention well, it must also suspend the vCPU at appropriate times. To determine *when* the vCPU should be suspended, *Gleaner* monitors the time spent in *sched_yield()* calls to determine whether the calls return immediately. If a *sched_yield()* call spends much more time than that needed by returning immediately without relinquishing the vCPU, the finish of the call denotes the beginning of an idling operation. For timeouts, *Gleaner* accumulates the time spent in consecutive *sched_yield()* calls that return immediately and compares the time against the time-out value to determine whether a time-out should be triggered. *Gleaner* sets the timeout value of the idling operation based on the cost incurred by context transitions if hardware resources are released to lower layers in the software stack, and calls *nanosleep(1)* to effect the actual vCPU suspension.

At user level, imbalanced scheduling can be achieved by changing the CPU affinity of application threads, but an easier yet more scalable way is to leverage the resource container support on the guest OS, e.g., cgroup support on Linux or zone support on Solaris. *Gleaner* creates a resource container for workload threads. It dynamically adjusts the vCPUs assigned to the container based on the policies described in Section 5. The workload threads will be accordingly redistributed by the guest OS scheduler on the assigned vCPUs upon every adjustment. For the vCPUs that are not assigned to the container, the *yielding threads* on them are put into sleep; thus these vCPUs will be suspended by the VMM to reclaim the physical resources.

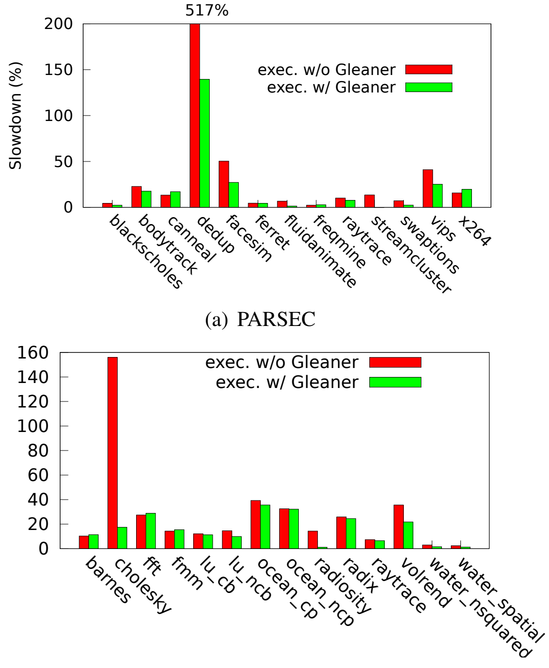
To get the measurements required by *imbalanced scheduling* (l_{comp}, g_{comp}, and l_{idle}) on active vCPUs, *Gleaner* periodically collects vCPU times spent by the yielding threads and workload threads (denoted by T_{yield} and T_{work}, respectively), as well as their number of context switches (S_{yield} and S_{work}, respectively). Then, l_{comp} is T_{work}/S_{yield}, g_{comp} is T_{work}/S_{work}, and l_{idle} is T_{idle}/S_{yield}.

7 Experiments

We evaluated our prototype implementation of *Gleaner* on a Dell™ PowerEdge™ R720 server with 64GB of DRAM and two 2.40GHz Intel® Xeon® E5-2665 processors, each of which has 8 cores. VMs were created with 16 virtual CPUs and 16GB of memory. The VMM is KVM [13], with EPT support and PLE support enabled. Both the host OS and the guest OS are Ubuntu version 12.04 with the Linux kernel version updated to 3.9.4. To prevent the performance degradation caused by CPU power management to latency sensitive applications, we disabled the C states deeper than C1 (including C1E) [28]. Please note that the change of these settings does not favor *Gleaner* and *Gleaner* can improve application performance by larger percentages when the C states are enabled.

We selected the benchmarks in PARSEC 3.0 and SPLASH 2X suites, SysBench [18], and matmul. We compiled the PARSEC and SPLASH2X benchmarks using gcc with the default settings of the *gcc-pthreads* configuration in PARSEC 3.0. We used the *parsecmgmt* tool in the PARSEC package to run them with native input and with the minimum number of threads set to 16 in the "-n" option. We used the OLTP test mode of SysBench to benchmark a MySQL database's performance on VMs with OLTP workloads.

We performed two sets of experiments. In the first set of experiments, we launch one VM on the system, and run the benchmarks in the VM. These experiments test the effectiveness of *Gleaner* on improving application performance in virtualized environment with dedicated physical resources. At the same time, we aim to confirm that the performance improvement is achieved without increasing energy consumption. In the second set of experiments, we launch two or more VMs and run one benchmark in each of the VMs. The VMs contend for hardware resources and test the capability of *Gleaner* to improve the performance of synchronization-intensive applications and overall system throughput on an over-subscribed system.



(a) PARSEC

(b) SPLASH2X

Figure 5: Virtualization slowdowns of PARSEC and SPLASH2X benchmarks on 16 cores with and without *Gleaner*.

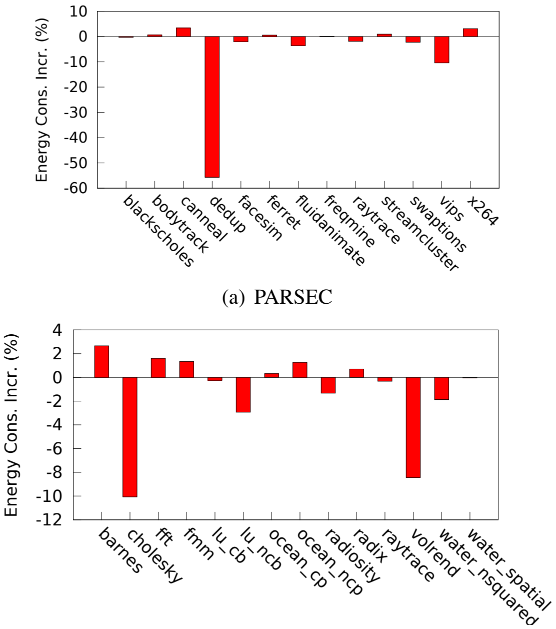
7.1 Single VM Experiments

We first execute the benchmarks in a single virtual machine running alone. We execute each benchmark in the following three scenarios: in the host OS, in the guest OS without Gleaner enabled, and in the guest OS with Gleaner enabled. In Figure 5, we report the slowdowns of the benchmark in the latter scenarios relative to its execution in the first scenario.

As shown in the figure, *Gleaner* is especially effective for the benchmarks suffering from the high blocking synchronization overhead. For example, virtualization slows down PARSEC's dedup and facesim benchmarks by 517% and 51%, respectively. With *Gleaner*, the slowdowns were reduced to 138% and 27%. The SPLASH2X benchmarks cholesky and volrend slow down by 155% and 35% without *Gleaner* but only by 19% and 21%, respectively, with it enabled.

For benchmarks that are not sensitive to blocking synchronization overhead, such as PARSEC's ferret and freqmine and SPLASH2X's water_nsquared and water_spatial, *Gleaner* neither improves nor degrades their performance, indicating that its overhead is very low.

For other benchmarks, *Gleaner* may slightly reduce or



(a) PARSEC

(b) SPLASH2X

Figure 6: Energy consumption increases of PARSEC and SPLASH2X benchmarks on 16 cores with *Gleaner*. Negative bars indicate energy consumption reduction.

increase their execution times. For example, the execution times of bodytrack and lu_ncb reduce by 6% and 5%, respectively, and *Gleaner* slightly increases the execution times of canneal (3%) and x264 (5%).

Gleaner may increase the execution times because, implemented at user level, it may not be able to quickly detect the increasing concurrency levels at the beginning of some new execution phases, and thus it cannot promptly adjust the number of active vCPUs to maximize throughput. The problem could be addressed with some assistance from the guest OS. For example, when threads are woken up (i.e., the concurrency level increases), the guest OS could notify *Gleaner* to increase the number of vCPUs.

On average, without *Gleaner*, the execution times of the PARSEC benchmarks and SPLASH2X benchmarks slow down by 55% and 30% when virtualized, relative to native execution, but with the tool enabled, the average slowdowns were reduced to 20% and 17%, respectively.

Gleaner uses yielding threads to keep some vCPUs busy even when they do not have any threads to run. This potentially increases energy consumption; however, the energy consumed by yielding threads can be justified if a reduction in execution time results, which in turn can be translated to reduced energy consumption. Moreover, *Gleaner* consolidates application threads and adjusts the number of active vCPUs to suppress the energy consumed by yielding threads. This significantly reduces the energy consumed by yielding threads.

To test whether or not *Gleaner* increases energy consumption, we used the IPMI OEM utility to measure the energy consumption of the system during the execution of each benchmark, and compared the energy consumption in the last two scenarios. The energy consumption increases are as shown in Figure 6. The data show that, although *Gleaner* may slightly increase the energy consumption for some benchmarks, it reduces energy consumption for many, especially for the benchmarks suffering from blocking synchronization overheads. Energy consumption is reduced primarily by reducing execution times: the benchmarks with larger execution time reductions usually show larger energy consumption reductions. The reductions in energy consumption are not proportional to reductions in execution time because yielding threads increase the power consumption when *Gleaner* is enabled. On average, *Gleaner* reduces the energy consumption by 5% for PARSEC benchmarks and by 1% for SPLASH2X benchmarks.

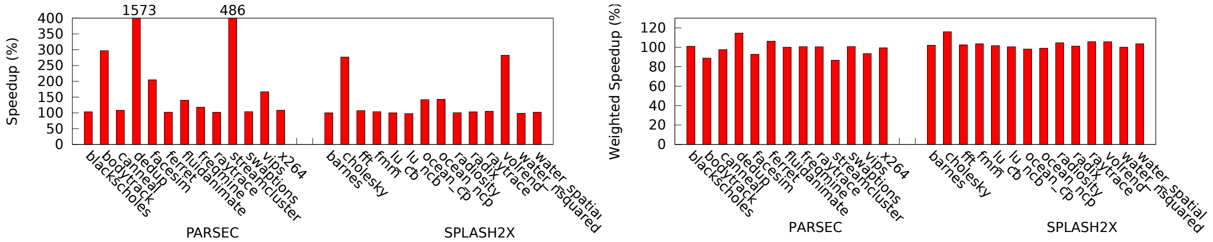
7.2 Experiments with Co-Running VMs

In this subsection, we present the experimental results when the system is oversubscribed. We launch two virtual machines, one with the PARSEC or SPLASH2X benchmark under test and one with matmul running repeatedly. As shown in Figure 1(b), synchronization-intensive benchmarks suffer much higher slowdowns than they do on a VM with dedicated hardware. With the first part of the experiments, we show that *Gleaner* can effectively speed up these applications on oversubscribed systems. Then, in the second part of the experiments, we show that *Gleaner* can improve system throughput by reducing the overhead caused by vCPU switches.

7.2.1 Reducing Application Execution Times

On an oversubscribed system, *Gleaner* improves the performance of synchronization-intensive applications by preventing hardware resources from being taken by other VMs if the resources are to be used soon. In the experiments, we use the application performance in the VM without *Gleaner* enabled as a baseline. In Figure 7(a), we show the speedups of the PARSEC and SPLASH2X benchmarks when *Gleaner* is enabled, relative to the baseline. In addition to the performance of individual applications, we also want to investigate how the idling operations affect system throughput. Thus, we use *Weighted-Speedup* to measure the system throughput, which is the average speedups of the benchmark and matmul, relative to their performance when *Gleaner* is disabled. The system throughput when *Gleaner* is disabled is always 1. Thus, in Figure 7(b), we only show the throughput of the system when *Gleaner* is enabled.

For the benchmarks that suffers significantly from



(a) speedup

(b) throughput

Figure 7: Speedup and system throughput of PARSEC and SPLASH2X benchmarks on a 16-vCPU VM with *Gleaner* on the oversubscribed system.

the overhead of blocking synchronization, e.g., dedup, streamcluster, bodytrack, cholesky, and volrend, *Gleaner* can dramatically improve their performance. It reduces the execution times by several factors (up to 16x for dedup). For benchmarks that are not sensitive to the overhead of blocking synchronization, e.g. blackscholes, ferret, and freqmine, *Gleaner* does not reduce their performance. For all the selected benchmarks, the average speedup is 203% when *Gleaner* is enabled.

The improvement of application performance does not come without cost. Idling operations may reduce efficiency because they prevent hardware resources from being utilized by other vCPUs. But they may also improve efficiency by reducing excessive costly vCPU switches. Therefore, we observed that *Gleaner* improves throughput for some workloads and reduces throughput for others. Generally, the system has similar throughputs when *Gleaner* is enabled (the average throughput is 2% higher than when *Gleaner* is disabled).

We notice that if the performance degradation of the benchmark is due to frequent barrier synchronization or condition variable synchronization, *Gleaner* usually reduces system throughput (e.g., bodytrack, streamcluster, and vips). This is because *Gleaner* usually cannot consolidate the threads of the application. The performance of such applications is more sensitive to the delay of its computation than other applications. The delay caused by *imbalanced scheduling* may degrade the application performance. For example, delaying the computation of one thread and making it the last one reaching a barrier will effectively delay all the other threads waiting at the barrier. In contrast, for the threads contending for a mutex, if the computation of a thread is delayed and it reaches a synchronization point late, the delay will not block other threads. When *imbalanced scheduling* is not used to reduce the length of idle periods, *Gleaner* cannot effectively minimize the cost of the idling operations handling these idle periods, which in turn reduces system throughput.

On the other hand, if the performance degradation of a benchmark is due to frequent mutex synchronization (e.g., dedup, ferret, and cholesky), *Gleaner* usually can increase the system throughput, albeit the gains are less than 15%. The reason for the modest gains is that, when the VM of these benchmarks run with the VM of matmul, the vCPU switches are not frequent, because the vCPU scheduling policy of KVM enforces a minimum time slice for the vCPUs running matmul. Thus, there is limited potential for *Gleaner* to improve performance.

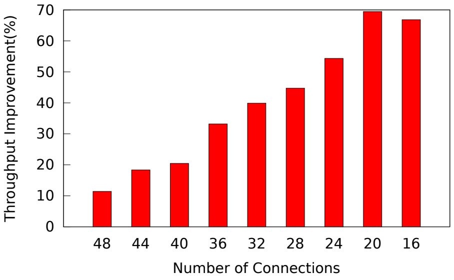


Figure 8: Throughput improvements of MySQL servers driven by the OLTP workload generated by SysBench.

We replaced matmul with freqsleep and repeated the above experiments to compare the performance of the benchmarks under these two settings. The benchmarks showed similar performance after replacing matmul with freqsleep. Even for dedup, which suffered the largest performance variation when *Gleaner* was not enabled (its execution time was 7x longer co-running with matmul than with freqsleep), after *Gleaner* was enabled its execution co-running with matmul was only 21% longer than with freqsleep. For the PARSEC and SPLASH2X benchmarks, with *Gleaner* enabled, the average execution time was 15% smaller when freqsleep replaces matmul, while the number is 3X with *Gleaner* disabled.

7.2.2 Improving System Throughput

With some commercial workloads and scientific workloads, frequent vCPU switches can significantly degrade system throughput. For such workloads, *Gleaner* can effectively reduce vCPU switches and improve system throughput. To demonstrate this capability of *Gleaner*, we select dedup and the SysBench OLTP benchmark, which was designed for MySQL server benchmarking.

First, we use SysBench to generate the OLTP workload to drive the MySQL database servers running in two VMs. The workload consists of a mixture of back-to-back transactions on a table with 1 million records as specified with SysBench OLTP "advanced transactional" test mode. In the experiment, we change the number of connections between SysBench and MySQL to vary the workload. Since each connection is backed by a MySQL server thread, when we reduce the number of connections, the number of server threads on each vCPU is also reduced; thus the chance for a vCPU becoming idle increases. Figure 8 shows that the throughput of MySQL servers is improved by increasingly larger percentages (upto 69%) by enabling *Gleaner* on the VMs running MySQL, when the number of connections is decreased from 48 to 16, with a peak at 20.

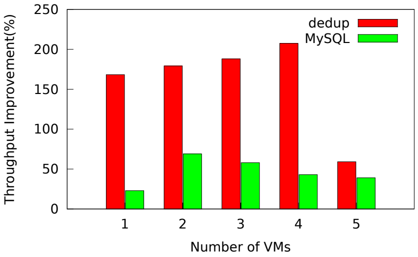


Figure 9: Throughput improvements of dedup and MySQL varying the number of VMs.

Finally, we vary the number of VMs running on the physical machine and test how the throughput improvements change with dedup and MySQL (the number of connections to each MySQL instance is 20). When the the number of VMs is increased to 5, the application performance is significantly degraded relative to that with one VM. For example, the average response time of the MySQL servers is increased by about 3X. Thus, we did not further increase the number of VMs. As shown in Figure 9, for all the above settings, *Gleaner* can substantially improve throughput. On average, it improves the throughput by 160% for dedup and 45% for MySQL database server.

We expect that the improvements increase with the number of VMs, because the contention for hardware resources increases with more VMs. As shown in Figure 9, dedup shows this trend before the number of VMs is smaller than 5. When the number of VMs is increased to 5, *Gleaner* cannot improve the throughput as much as it does with fewer VMs due to memory overcommitment. For the OLTP workload generated by SysBench, we observe that the throughput improvement gradually reduces when the number of VMs increases from 2 to 5. This is because when more VMs share the same physical cores, each VM has fewer active vCPUs, which in turn alleviates the mutex contention in MySQL servers, making the server threads less likely to be blocked.

8 Related Work

While there are a number of studies identifying performance overheads of virtualized execution [1, 6, 7, 8, 14, 15, 16, 17, 25, 26], most of them focus on the overhead incurred by I/O operations and spinlock synchronization. To reduce the performance degradation caused by spinlock synchronization in virtual machines, a few approaches have been proposed, including vCPU scheduling approaches [5, 6, 12, 20, 23, 24, 26], hardware approaches [21, 29], and improved spinlock design [19]. Flex also addresses the fairness issue of scheduling multicore VMs [20]. None of these studies identify and address the performance degradations caused by blocking synchronizations in multicore virtualized environments (the BWW problem).

The blocked-waiter wakeup problem was also described in a technical report [22] and a workaround was proposed to run a user-level idle daemon to avoid halting vCPUs, which is similar to yielding threads in our solution. The workaround helps improve performance for VMs with dedicated hardware (at the cost of increased energy consumption). But it causes performance degradation when the system is oversubscribed, similar to that caused by the LHP problem. This paper systematically analyzes the issues with the BWW problem and provides an efficient and universal solution.

The BWW problem, as well as the LHP problem, is caused by the lack of coordination between the vCPU scheduler in the VMM and the task scheduler in the guest OS. Thus, one solution is to enforce the collaboration between the schedulers using techniques similar to scheduler activations [3]. However, this approach requires intensive modifications to both the VMM and the guest OS. Another approach is to minimize vCPU scheduling by assigning one runnable vCPU to each pCPU and making other vCPUs offline to computation in guest OSes [23]. This approach avoids vCPU preemption and can effectively address the LHP problem. Making some vCPUs offline may be able to reduce the idle time on the online vCPUs and reduce the chance for them to become idle. But it may not be effective for the BWW problem, because every vCPU state transition from busy to idle still incurs a switch between vCPUs and the number of online vCPUs in each individual VM may not be adjusted in a way to reduce idle time.

Retaining idle resources for anticipated usages is a common scheduling technique in system designs [9, 10, 11]. It avoids the high overhead associated with resource reallocation and state switches. At a high level, the resource retention techniques in *Gleaner* share a similar idea with these designs.

A preliminary version of this paper appeared in HotCloud'13 [4].

9 Conclusion and Future Work

This paper identifies an understudied problem for running multithreaded applications in virtualized multicore environments. Namely, the costs incurred by blocking synchronization in virtualized environments can exact a significant performance penalty when scaling multicore applications to take advantage of larger and larger core counts. This paper proposes and designs *Gleaner* as a solution, which combines resource retention approaches with idling operations and consolidation scheduling. Extensive experiments show that *Gleaner* can significantly improve application performance and system throughput in virtualized environments. It can also reduce application performance variations when multiple VMs share the same physical resources.

As future work, we want to test and extend our approach to reduce the overhead due to vCPU state transitions caused by operations other than blocking synchronization. For example, SSD access latencies are typically tens of microseconds in Amazon EC2 instances. The state transitions of vCPUs when they are waiting for I/O can double the I/O latency and reduce I/O throughput at the same time. It seems that *Gleaner* could be adapted to reduce this extra cost incurred by vCPU state transitions.

Acknowledgments. We thank the anonymous reviewers for their constructive comments, and Dr. Haibo Chen for his helpful suggestions as the shepherd for this paper. This research was supported in part by the Intel Science and Technology Center for Cloud Computing.

References

[1] ADAMS, K., AND AGESEN, O. A comparison of software and hardware techniques for x86 virtualization. In *ACM ASPLOS 2006*, pp. 2–13.

[2] AMAZON, 2014. http://aws.amazon.com/ec2/instance-types/instance-details/.

[3] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM SOSP 1991*, pp. 95–109.

[4] DING, X., GIBBONS, P. B., AND KOZUCH, M. A. A hidden cost of virtualization when scaling multicore applications. In *HotCloud* (2013).

[5] DRUMMONDS. Co-scheduling SMP VMs in VMware ESX server, 2008. http://communities.vmware.com/docs/DOC-4960.

[6] FRIEBEL, T., AND BIEMUELLER, S. How to deal with lock holder preemption. *Xen Summit North America* (2008).

[7] GORDON, A., AMIT, N., HAR'EL, N., BEN-YEHUDA, M., LANDAU, A., SCHUSTER, A., AND TSAFRIR, D. ELI: bare-metal performance for I/O virtualization. In *ACM ASPLOS 2012*, pp. 411–422.

[8] HAN, J., AHN, J., KIM, C., KWON, Y., CHOI, Y.-R., AND HUH, J. The effect of multi-core on HPC applications in virtualized systems. In *Euro-Par 2010*, pp. 615–623.

[9] IRANI, S., SHUKLA, S., AND GUPTA, R. Algorithms for power savings. In *ACM-SIAM SODA 2003*, pp. 37–46.

[10] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *ACM SOSP 2001*, pp. 117–130.

[11] KARLIN, A. R., LI, K., MANASSE, M. S., AND OWICKI, S. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *ACM SOSP 1991*, pp. 41–55.

[12] KIM, H., KIM, S., JEONG, J., LEE, J., AND MAENG, S. Demand-based coordinated scheduling for SMP VMs. In *ACM ASPLOS 2013*, pp. 369–380.

[13] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. KVM: the linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), pp. 225–230.

[14] LANDAU, A., BEN-YEHUDA, M., AND GORDON, A. SplitX: split guest/hypervisor execution on multi-core. In *USENIX WIOV 2011*, pp. 1–7.

[15] LANGE, J. R., PEDRETTI, K., DINDA, P., BRIDGES, P. G., BAE, C., SOLTERO, P., AND MERRITT, A. Minimal-overhead virtualization of a large scale supercomputer. In *ACM VEE 2011*, pp. 169–180.

[16] LUSZCZEK, P., MEEK, E., MOORE, S., TERPSTRA, D., WEAVER, V. M., AND DONGARRA, J. Evaluation of the HPC challenge benchmarks in virtualized environments. In *Euro-Par 2011*, pp. 436–445.

[17] MENON, A., SANTOS, J. R., TURNER, Y., JANAKIRAMAN, G. J., AND ZWAENEPOEL, W. Diagnosing performance overheads in the Xen virtual machine environment. In *ACM VEE 2005*, pp. 13–23.

[18] MYSQL AB. SysBench manual, 2010.

[19] OUYANG, J., AND LANGE, J. R. Preemptable ticket spinlocks: Improving consolidated performance in the cloud. In *ACM VEE 2013*, pp. 191–200.

[20] RAO, J., AND ZHOU, X. Towards fair and efficient smp virtual machine scheduling. In *ACM PPoPP 2014*, pp. 273–286.

[21] RIGHINI, M. Enabling Intel® virtualization technology features and benefits. Tech. rep., Intel, 2010.

[22] SONG, X., CHEN, H., ZANG, B., SONG, X., CHEN, H., AND ZANG, B. Characterizing the performance and scalability of many-core applications on virtualized platforms. Tech. Rep. FDUPPITR-2010-002, Parallel Processing Institute, Fudan University, 2010.

[23] SONG, X., SHI, J., CHEN, H., AND ZANG, B. Schedule processes, not VCPUs. In *APSys 2013*, pp. 1:1–1:7.

[24] SUKWONG, O., AND KIM, H. S. Is co-scheduling too expensive for SMP VMs? In *EuroSys 2011* (2011), ACM, pp. 257–272.

[25] TICKOO, O., IYER, R., ILLIKKAL, R., AND NEWELL, D. Modeling virtual machine performance: Challenges and approaches. *SIGMETRICS Perform. Eval. Rev. 37*, 3 (Jan. 2010), 55–60.

[26] UHLIG, V., LEVASSEUR, J., SKOGLUND, E., AND DANNOWSKI, U. Towards scalable multiprocessor virtual machines. In *VM 2004*, pp. 43–56.

[27] VMWARE. Host power management in VMware vSphere 5, 2010. http://www.vmware.com/files/pdf/hpm-perf-vsphere5.pdf.

[28] VMWARE, 2013. http://www.vmware.com/resources/techresources/10205.

[29] WELLS, P. M., CHAKRABORTY, K., AND SOHI, G. S. Hardware support for spin management in overcommitted virtual machines. In *PACT 2006*, ACM, pp. 124–133.

HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management

Yangchun Fu Junyuan Zeng Zhiqiang Lin
The University of Texas at Dallas
{firstname.lastname}@utdallas.edu

Abstract

To direct the operation of a computer, we often use a shell, a user interface that provides accesses to the OS kernel services. Traditionally, shells are designed atop an OS kernel. In this paper, we show that a shell can also be designed below an OS. More specifically, we present HYPERSHELL, a practical hypervisor layer guest OS shell that has all of the functionality of a traditional shell, but offers better automation, uniformity and centralized management. This will be particularly useful for cloud and data center providers to manage the running VMs in a large scale. To overcome the semantic gap challenge, we introduce a reverse system call abstraction, and we show that this abstraction can significantly relieve the painful process of developing software below an OS. More importantly, we also show that this abstraction can be implemented transparently. As such, many of the legacy guest OS management utilities can be directly reused in HYPERSHELL without any modification. Our evaluation with over one hundred management utilities demonstrates that HYPERSHELL has 2.73X slowdown on average compared to their native in-VM execution, and has less than 5% overhead to the guest OS kernel.

1 Introduction

With the increasing use of cloud computing and data centers today, there is a pressing need to manage a guest operating system (OS) directly from the hypervisor layer. For instance, when migrating a virtual machine (VM) from one place to another, we would like to directly configure its IP address without logging into the system (if that is possible), similarly for firewall rule update; when there is a malicious process detected, we would like to directly kill it at hypervisor layer, similarly for malicious kernel modules; when there is a need to scan viruses, we would like to uniformly scan viruses for all of the running VMs regardless of who owns and manages the VM, whether the VMs might be using an unknown file system, or whether the file systems might be encrypted.

However, if we use a traditional OS shell, a user interface that is automatically executed when a user successfully logs in a computer, this would first require an administrator's password. But, hypervisor providers may not (always) have the administrator's password for each VM, and even when they do have the passwords, it is painful to maintain them considering today large cloud providers usually run millions of VMs (e.g., there were over one million VMs running in Skytap cloud as of January 2012 [7]). Second, it would also require the installation of the management utilities inside each guest OS. Whenever there are updates for these utilities, it is painstaking to update all of them in each VM. Therefore, the presence of a hypervisor layer shell (HYPERSHELL for brevity) for all guest OS would allow cloud providers to have an automated, uniformed, and centralized service for in-VM management.

Unfortunately, such a layer below shell is challenging to implement because of the semantic gap [9]. Specifically, the semantic gap exists since at the hypervisor layer we have access only to the zeros and ones of the hardware level state of a VM—namely its CPU registers and physical memory. But what a hypervisor layer program wants is the semantic information about the guest OS, such as the running processes, opened files, live network connections, host names, and IP addresses. Therefore, a layer below management program must reconstruct the guest OS abstractions in order to obtain meaningful information. A typical approach to do so is to traverse the kernel data structure, but such an approach often requires a significant amount of manual effort.

To advance the state-of-the-art, we introduce a new abstraction called Reverse System Call (R-syscall in short) to bridge the semantic gap for hypervisor layer programs that will be executed in our HYPERSHELL. Unlike traditional system calls that serve as the interface for application programs from a layer below, R-syscall serves as the interface in a reverse direction from a layer up (with a way similar to an upcall [11]). While hypervisor programmers can use our R-syscall abstraction to develop new guest OS management utilities, to largely reuse the existing legacy software (e.g., ps/lsmod/netstat/ls/cp) we make the system call interface of R-syscall transparent to the legacy software, resulting in no modification when using them in HYPERSHELL. In addition, we also make HYPERSHELL transparent to the guest OS, and we do not modify any guest OS code. All of our design and implementation is done at the hypervisor layer.

We have implemented HYPERSHELL. We show that by using the abstraction of R-syscall, we can quickly have a large number of hypervisor layer guest OS management utilities by reusing the existing legacy software (due to its transparency). In our current evaluation, we have tested with over 100 common system administrative utilities. All of them can be correctly executed in HYPERSHELL. The average performance overhead for these utilities is 2.73X slowdown compared to their native in-VM execution. Both micro and macro benchmark evaluation shows that HYPERSHELL has very small overhead (less than 5%) to the guest OS kernel.

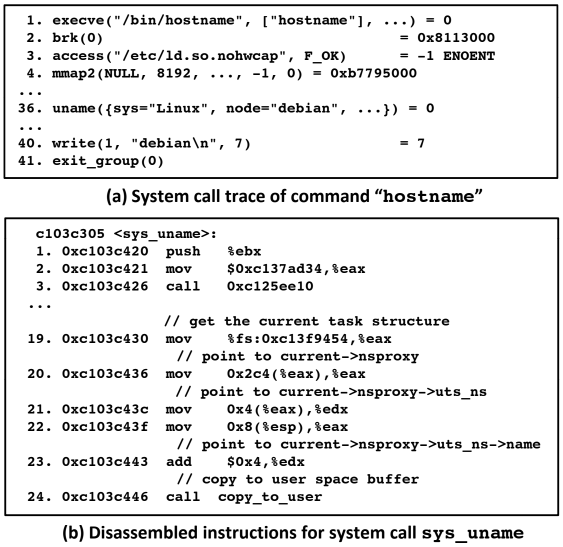
In short, this paper makes the following contributions:

- We present HYPERSHELL, a new hypervisor layer shell for automated guest OS management, without using any user accounts from a guest OS.
- We introduce an R-syscall abstraction that allows hypervisor programmers to develop guest OS management utilities without worrying about the semantic gap. Its transparency feature also directly allows many of the legacy utilities to be executed in HYPERSHELL without any modification.
- We have implemented the whole system. We show that HYPERSHELL is practical, and can be used for timely, uniformed, and centralized guest OS management, especially for private cloud.

2 Background and Overview

Challenges. HYPERSHELL aims at executing guest OS management utilities at the hypervisor layer with the same effect as executing them inside an OS. To this end, we are facing two major challenges:

- **How to bridge the semantic gap**. In HYPERSHELL, guest OS management utilities execute below an OS kernel. However, for OS below software, there are no OS abstractions. For example, there is no pid, no FILE, and no socket. Therefore, we have to reconstruct these abstractions such that the utility software understands the guest OS states and can perform the management.
- **How to develop the utilities**. Suppose we have a perfect approach to bridging the semantic gap, we still have to develop the guest OS management software. Should we develop the software from scratch, or can we reuse any legacy (binary or source) code? Ideally, we would like to reuse the existing binary code as there are already lots of OS management utilities, and we show that this approach is feasible.



```
 1. execve("/bin/hostname", ["hostname"], ...) = 0
 2. brk(0)                                   = 0x8113000
 3. access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
 4. mmap2(NULL, 8192, ..., -1, 0) = 0xb7795000
...
36. uname({sys="Linux", node="debian", ...}) = 0
...
40. write(1, "debian\n", 7)                 = 7
41. exit_group(0)
```

(a) System call trace of command "hostname"

```
  c103c305 <sys_uname>:
  1. 0xc103c420  push   %ebx
  2. 0xc103c421  mov    $0xc137ad34,%eax
  3. 0xc103c426  call   0xc125ee10
 ...
                // get the current task structure
 19. 0xc103c430  mov    %fs:0xc13f9454,%eax
                 // point to current->nsproxy
 20. 0xc103c436  mov    0x2c4(%eax),%eax
                 // point to current->nsproxy->uts_ns
 21. 0xc103c43c  mov    0x4(%eax),%edx
 22. 0xc103c43f  mov    0x8(%esp),%eax
                 // point to current->nsproxy->uts_ns->name
 23. 0xc103c443  add    $0x4,%edx
                 // copy to user space buffer
 24. 0xc103c446  call  copy_to_user
```

(b) Disassembled instructions for system call sys_uname

Figure 1: System call trace of utility hostname and one of its sys_uname implementation.

Key Insights. Before describing how we solve these challenges, we would like to first revisit how an in-VM management utility executes. Suppose we want to know the host name of a running OS, we can use utility software such as hostname to fulfill this task. In particular, as illustrated in Fig.1(a), it will execute 41 system calls (syscall for short) in Linux kernel 2.6.32.8, our testing guest kernel. Among these syscalls, sys_uname is the one that really returns the host name. Also, as shown in Fig.1(b), this syscall will traverse the current task structure and dereference the field current->nsproxy->uts_ns->name to eventually retrieve the machine name.

If we implement the same hostname utility and execute it in HYPERSHELL, and if we use a manual approach to bridging the semantic gap, we have to traverse the data structure again, in the same way as how sys_uname does. Since the only interface for user level programs to request OS kernel services is through syscall, and the execution of a syscall is often trusted, then why not let hypervisor programs directly use the syscall abstractions provided by the guest OS? As such, we do not have to develop any code to reconstruct the guest OS abstractions. This is one of the key insights of designing HYPERSHELL.

Another key insight is that not all the syscalls should be executed inside the guest OS. One example is the write syscall that prints the "host name" to the screen. If we execute it inside the guest OS, we would not be able to observe the output from HYPERSHELL. Therefore, we introduce an R-syscall abstraction that is used by hypervisor programmers to annotate the syscalls that need to be redirected and executed inside the guest OS.

In addition, while hypervisor programmers can use our R-syscall abstraction to develop new software to manage

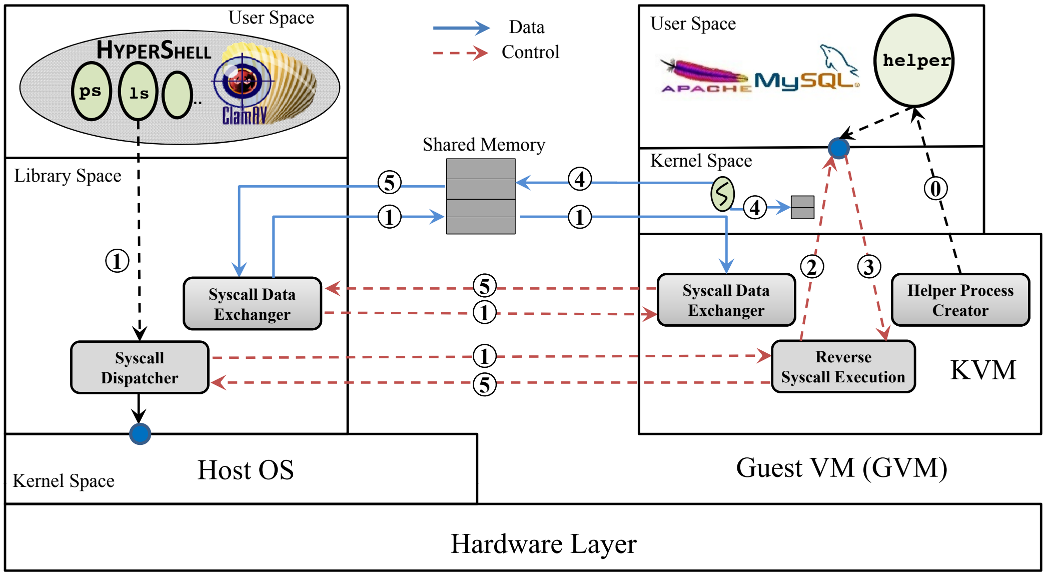


Figure 2: An Overview of the HYPERSHELL Design.

the guest OS, there are already lots of legacy utilities running inside a VM for the same purposes. For instance, there are over hundreds of tools in core-utility, util-linux, and net-tools for Linux OS. If we can make our R-syscall transparent to the legacy software, then there is no need to annotate the R-syscall and we can directly execute the legacy software in HYPERSHELL. For instance, in `hostname` example, only the `uname` syscall needs this abstraction. We can thus hook the execution of the syscall and use a transparent policy to determine whether a given syscall is an R-syscall.

Scope and Assumptions. As HYPERSHELL is executed at the hypervisor layer and will also invoke the syscalls from the guest OS, we assume everything below is trusted. This includes the guest OS kernel, host OS and the hypervisor code. Ensuring the hypervisor and guest kernel integrity is an independent problem and out of scope of this paper. In fact, recently there have been many efforts aiming at ensuring the guest kernel and hypervisor integrity (e.g., SecVisor [33] and HyperSafe [37]). Also note that HYPERSHELL is designed mainly for automated guest OS management and not for security. While it could defeat certain attacks such as guest user level viruses, it cannot defend against any guest kernel level attacks.

To make our discussion more focused, we assume a guest OS running with a 32-bit Linux kernel atop x86 architecture. For the hypervisor, we focus on the design and implementation of HYPERSHELL using KVM [4].

Overview. An overview of our HYPERSHELL is presented in Fig. 2. For a KVM based virtualization system, there are two kinds of OSes: one is the guest OS that is executed atop a KVM hypervisor, and the other is the host OS that hides the underlying hardware resources and provides the virtualized resources to KVM. The goal of HYPERSHELL is to execute the guest OS management utilities from the host OS to manage the guest OS. To this end, there are five key components: two located inside the library space of the host OS, and three located at the hypervisor layer of the GVM.

To use HYPERSHELL, assume hypervisor managers use `ls` (or other utilities such as `ps` or `hostname`) to list the guest files in a given directory. To get started, they will launch `ls` in our host OS. The real execution of `ls` will be divided into a master process that is executed inside the host OS, and a helper process that is executed in the GVM. Only when an R-syscall gets executed will we forward the execution of this syscall to a helper process in the GVM and map the execution result (e.g., the directory entries) back such that `ls` can continue its execution in the master process. There are five key steps involved during the execution of an R-syscall:

- **Step** ①: Right after a syscall enters the library space in the host OS, our Syscall Dispatcher intercepts it. If it is not an R-syscall, it directly traps to the host OS kernel for the execution. Otherwise it fetches the syscall number and arguments, and invokes our Syscall Data Exchanger at the host OS side which communicates with its peers at the GVM side with the detailed syscall execution information. Next, our master process gets paused and will be resumed at **Step** ⑤ when the redirected R-syscall finishes the execution. At the GVM side, according to each specific syscall specification, the Syscall Data Exchanger will set up the corresponding memory state for the to-be-executed R-syscall.

- **Step** ②: Our Reverse Syscall Execution will wait until the helper process traps to kernel. The helper process is created at **Step** ⓪ right after the execution of the management utilities in HYPERSHELL, or can be executed as a daemon depending on the settings.

- **Step** ③: Our Reverse Syscall Execution directly injects the execution of the R-syscall with the corresponding arguments and memory mapping, and makes the R-syscall be executed under the helper process kernel context. Note that such an R-syscall injection and execution mechanism works similar to function call injection from debuggers but with a more powerful capability because of the layer below control from hypervisor.

- **Step** ④: During the execution of the R-syscall, if there is any kernel state update to the guest OS, this syscall will directly update the kernel memory as usual (e.g., `sysctl` that changes the kernel configuration). If there is any user space update (such as the `buffer` in `read` syscall), it directly updates to the shared memory created by the Syscall Data Exchanger in **Step** ①.

- **Step** ⑤: Right after the execution of the syscall exit of the R-syscall, we notify Syscall Dispatcher and Syscall Data Exchanger at the host OS side. We also copy the data from the shared memory to the user space of the master process, if the R-syscall has any memory update. We also resume the execution of the master process and directly return to its user space for continued execution. Regarding the helper process state in the GVM: if the master process terminates, it will also be terminated (in non-daemon mode); otherwise, it will keep executing `int3`[1] such that our Reverse Syscall Execution can always take control of the helper process from the hypervisor layer.

3 Host OS Side Design

3.1 Syscall Dispatcher

The key idea of HYPERSHELL in bridging the semantic gap is to selectively redirect and execute a syscall in the guest OS. (The selected one is called an R-syscall). As shown in Fig.1(a), not all the syscalls belong to R-syscalls. Therefore, the first step in our Syscall Dispatcher design is to systematically examine all of the Linux syscalls and define our reverse execution policy for each syscall.

Syscall Execution Policy. In our testing guest kernel Linux 2.6.32.8, there are 336 syscalls in total. Among them, we find that technically, nearly all of them can be redirected to execute in a guest OS. However, for process creation (e.g., `execve`, `fork`, `exit_group`), dynamic loading (e.g., `open`, `stat`, `read` when loading a shared library), memory allocation (e.g., `brk`, `mmap2`), and screen output (e.g., `write`), we would like them to be executed in the master process created in our HYPERSHELL.

<table>
<tr><th>The Syscall Trace of "<code>cp /etc/shadow /outside/shadow</code>"</th><th>Host OS</th><th>GVM</th></tr>
<tr><td><code>execve("/bin/cp",["cp","/etc/shadow","/tmp/shadow"],…= 0</code></td><td>✔</td><td></td></tr>
<tr><td><code>brk(0)   = 0x8824000</code></td><td>✔</td><td></td></tr>
<tr><td><code>access("/etc/ld.so.nohwcap", F_OK)   = -1 ENOENT</code></td><td>✔</td><td></td></tr>
<tr><td>…</td><td>✔</td><td></td></tr>
<tr><td><code>stat64('/etc/shadow',{st_mode=S_IFREG|0640,st_size=713, ...})=0</code></td><td></td><td>✔</td></tr>
<tr><td><code>stat64("/outside/shadow", 0xbf9bad78)   = -1 ENOENT</code></td><td>✔</td><td></td></tr>
<tr><td><code>open("/etc/shadow", O_RDONLY|O_LARGEFILE) = 0</code></td><td></td><td>✔</td></tr>
<tr><td><code>fstat64(0, {st_mode=S_IFREG|0640, st_size=713, ...}) = 0</code></td><td></td><td>✔</td></tr>
<tr><td><code>open("/outside/shadow", O_WRONLY|O_CREAT|…|O_LARGEFILE, 0640)=3</code></td><td>✔</td><td></td></tr>
<tr><td><code>fstat64(3, {st_mode=S_IFREG|0640, st_size=0, ...}) = 0</code></td><td>✔</td><td></td></tr>
<tr><td><code>read(0, "root::15799:0:99999:7:::\ndaemon:"..., 32768) = 713</code></td><td></td><td>✔</td></tr>
<tr><td><code>write(3, "root::15799:0:99999:7:::\ndaemon:"..., 713) = 713</code></td><td>✔</td><td></td></tr>
<tr><td><code>read(0, "", 32768)   = 0</code></td><td></td><td>✔</td></tr>
<tr><td><code>close(0)</code></td><td></td><td>✔</td></tr>
<tr><td><code>close(3)</code></td><td>✔</td><td></td></tr>
</table>

Table 1: Syscalls in `cp` with different execution policy.

Unfortunately, for the rest syscalls, it is also not always clear which syscalls need to be executed in the helper process. For instance, as shown in Table 1, suppose we want to copy `/etc/shadow` from the guest OS to the host OS; in this case, some of the file system related syscalls (e.g., `open/stat64/read/close`) are executed in the GVM, and some (e.g., `open/stat64/write/close`) are executed in HYPERSHELL. Even though we could leave the solution to hypervisor programmers, where they would specify which syscall needs to be executed in the master process or helper process, we would prefer to make an automated policy for these syscalls in order to allow for transparent reuse of the legacy binary code.

In general, syscalls are relatively independent of each other (e.g., `getpid` will just return a process ID, and `uname` will just return the host name). After having examined all of the 336 syscalls, we realize that the syscalls that have connections are often file system and `socket` related (e.g.,`open/stat64/read/write/close`), and these syscalls have dependences with the file descriptors. For instance, as illustrated in Table 1, if we can differentiate the file descriptor from the GVM and the host OS automatically, we can then transparently execute the existing legacy utility in HYPERSHELL without any modification.

Intuitively, we would use dynamic taint analysis [29] to differentiate the file descriptors that are accessed inside the GVM or the host OS. However, such a design would require instruction level instrumentation, which is often very slow. In fact, our earlier design adopted such a taint analysis approach by running HYPERSHELL in an emulator. Surprisingly, we have a new observation and we can actually eliminate the expensive dynamic taint analysis.

[1] An interrupt that is often used by debuggers to set up break points.

In particular, as a file descriptor is just an index (a 32-bit unsigned integer) to the opened files (and network `socket`) inside the OS kernel for each process, it has a limited maximum value (due to the resource constraints). In our testing Linux kernel, it is 1023 (which means a process can only open 1024 files at the same time). Also, it is extremely rare to perform data arithmetic operations on a file descriptor. Therefore, we can in fact add a distinctive value (e.g., 4096 or 8192) to the file descriptor returned by the GVM. Whenever such a descriptor is used by the GVM again, we subtract our added value. As such, we can differentiate whether a file descriptor is from the host OS or the GVM by simply looking at its value.

Whether a file descriptor should be returned from the GVM or the host OS depends on the semantics of `open`. Specifically, if it is opening the guest OS files (we can differentiate this based the parameters, and internally we add a prefix associated with the guest files), it is executed in the GVM; otherwise it is executed in the host OS. For instance, we know "/etc/shadow" is in the GVM, and "/outside/shadow" is in the host OS while executing "`cp /etc/shadow /outside/shadow`". Similarly, we can also infer the files involved in "`cp -R <src> <dst>`" by their names and their opening mode.

Syscalls in Dynamic Loader. To intercept the syscall, we use dynamic library interposition [13] (a technique that has been widely used in many applications such as LibSafe [36]). Interestingly, we notice that the syscalls executed in dynamic loader cannot be trapped by our library interposition. Therefore, syscalls executed while loading a dynamic library (e.g., `access/open/stat64/read/close`) will not be checked against our policy, and they will be executed directly on the host OS side, which is exactly what we want.

Summary. By default, the majority of the syscalls will be treated as redirectable and they will be executed in the GVM, except process execution and memory management related syscalls that will be executed in the host OS. All file system and network connection related syscalls will be checked against the file descriptor. Whether a file descriptor needs to be checked is determined by the semantics of the corresponding file operations.

3.2 Syscall Data Exchanger

Since we need to make an R-syscall executed in the GVM, we must inform the GVM with the corresponding context and also update the corresponding memory state at the host OS side to reflect the R-syscall's execution. Our Syscall Data Exchanger is designed for this goal.

Specifically, right after an R-syscall enters the library space (**Step** ①), we will retrieve the syscall arguments (e.g., the buffer address and size information) based on the corresponding syscall's specification. Then, we will inform its peer (to be discussed in §4.3) to prepare for the necessary arguments at the GVM side. Once an R-syscall finishes the execution (**Step** ⑤), we will pull the data back from the GVM to the host OS. All of these operations are quite straightforward.

4 Guest VM Side Design

4.1 Helper Process Creator

An R-syscall must be executed under a certain process execution context in the GVM. While we could hijack an existing process to execute an R-syscall, such an approach is too intrusive to the hijacked process. Therefore, we choose to create a helper process dedicated to executing our R-syscall in the guest OS. Regarding the permission of this helper process, it should have the highest privilege; otherwise an R-syscall may fail due to certain permissions. Also, it would terminate when the master process terminates (to minimize the impacts to the guest OS workloads). To have better performance while executing the management utilities in HyperShell, we can also have an option of creating a daemon process as the creation of a helper process takes additional time. There are only three instructions for this helper process as shown below:

```
00000001  cd 80        int 0x80
                  _loop:
00000003  cc           int 3
00000004  eb fd        jmp  _loop
```

Basically, it keeps executing `int3` (i.e., `while(1) int3`) with a prefix of `int 0x80`. We will explain why we use such an instruction sequence in §4.2.

Then the challenge lies in how to select a high privilege process to `fork` the helper process. Since all Linux kernels have an `init` process with PID 1, one option is to traverse the `pid` field of the `task_struct` for each process. But such a design would make HyperShell too OS-specific. Fortunately, since we are able to inject an R-syscall (discussed in §4.2), we are certainly able to inject `getpid` to inspect the return values. If it is 1, we can therefore infer that the current execution context is the `init` process, and we can then inject a `fork` syscall to create our helper process. Meanwhile, we will retrieve the child PID from the return value of `fork`, and then use `getpid` again to identity the helper process. Once we have identified it, we will pull its CR3 such that the hypervisor knows it is the `int3` that occurs in our helper process, not others (e.g., `gdb`) by looking at the CR3 value.

Consequently, we must design a mechanism to intercept the entry point and exit point of the syscall execution for each process in order to select the `init` process. Once we have created our helper process, we will not need this interception. We call the selection of `init` process *redirection initialization phase* (i.e., the RI-Phase that only occurs at **Step** ⓪) in the GVM. With hardware-assisted virtualization, we can rely on hardware mechanisms to intercept the execution of the syscall instructions. Ether [14], built atop the Xen hypervisor, leverages a page fault exception to capture syscall entry and syscall exit points. Nitro [31], based on the KVM hypervisor, leverages invalid segment exceptions to intercept the pair of `sysenter/int0x80` and `sysexit` syscalls for a single process. In our design, we extend Nitro to intercept the system wide syscall entry and exit pairs (for all processes).

4.2 Reverse Syscall Execution

After we have passed the RI-Phase, we are then ready to execute an R-syscall if there is any. Yet we have to solve two additional challenges: when and how to execute an R-syscall under our helper process context.

When to Inject a Syscall. At a given time, a process either executes in user space or kernel space. To trap to kernel, a process must use a syscall or interrupt (including exceptions). As an interrupt or exception can occur at arbitrary time, the OS must be designed in such a way that it is safe to trap to OS kernel and execute syscall or interrupt handler services at any time in user space. However, we cannot inject a syscall execution at arbitrary time in kernel space. This is because: (1) the injected syscall might make kernel state inconsistent. For instance, we might inject a syscall when the kernel is handling an interrupt, and there might be some synchronization primitives involved (e.g., `spin_lock`). After we inject a new syscall, if this syscall execution also happens to lock some data or release certain locks, it may cause inconsistency among these locks. (2) Similarly, we might make the non-interruptible code interruptible. For instance, if the kernel is executing the `cli` code block and has not executed `sti` yet, and if we inject a new syscall, this may make the non-interruptible code interruptible. (3) We might also overflow the kernel stack of a running process if it already has a large amount of data.

Therefore, to inject the execution of a syscall, we use the approach that right before entering the kernel space (e.g., `sysenter/int0x80`), or right after exiting to the user space of a running process, we will save the current execution context (namely all the CPU registers), and then execute the injected syscall (such as our `getpid` case in §4.1).

Regarding our helper process, we have a slightly different strategy to inject the R-syscall. In particular, when the `int3` traps to hypervisor, we change the current user level EIP (pointing to `cc` at this moment) to EIP-2, which points to "`int 0x80`"; meanwhile, we prepare for the necessary arguments such as setting up the corresponding registers. Then when control returns to the user space of the helper process, it will automatically execute the syscall we prepared for because we have changed its EIP. The use of `int3` is to make the control flow of the helper process trap to the hypervisor. There are also alternative approaches such as using a `cpuid` instruction.

How to Execute an R-syscall. To execute an R-syscall, we have to set up the syscall arguments and map the memory that will be used during the R-syscall execution. This is done by our Syscall Data Exchanger (§4.3) at **Step** ①. After that, the syscall will be executed as usual in the GVM. If there is any memory update to the user space, it will directly (**Step** ④) update to the shared memory that is allocated by our Syscall Data Exchanger. For kernel space, it directly updates the guest kernel. Once an R-syscall finishes, we inform the Syscall Dispatcher at **Step** ⑤, and push the updated memory back to the master process. At the GVM side, the helper process continues its execution of `int3`. When the master process exits, we terminate the helper process if it is not executed in the daemon mode.

4.3 Syscall Data Exchanger

As discussed in §3.2, we need to pass the corresponding syscall parameters to the GVM. Also, we need to map the data back to the host OS if there is any memory update. The Syscall Data Exchanger at the GVM side is exactly designed to achieve these goals.

One issue we have to solve is the virtual address relocation. This is because the same virtual addresses used by the host OS may not be available for the helper process in the GVM, and we have to relocate the virtual addresses used in the syscalls of the master process to the available addresses of the helper process. To this end, before the execution of the first R-syscall, we will first allocate a large buffer (as a cache) with a default size of 64K bytes by injecting a `mmap` syscall and recording the mapped virtual address of this buffer, denoted as V_g, and its size, denoted as S_g. (Certainly, the guest OS will automatically `munmap` this allocated space once the helper process terminates.) Then whenever there is an R-syscall (e.g., `read`) that has an argument with virtual address V_h and size S_h, we will use V_g as the buffer starting address instead of V_h, and if S_h is greater than S_g, we will inject `mmap` to map more caches.

Also, to avoid too many data transmissions between the host OS and the GVM, we allocate a shared memory between them. Right after the execution of the `mmap` syscall to allocate new pages for the redirected syscall, in the hypervisor layer we map the pages of the shared memory to the virtual address of the `mmap` returned page by traversing the page tables (rooted by the captured CR3) of the helper process, such that we do not have to perform an additional memory copy from the GVM to the shared memory. To prevent being swapped by the guest OS, we inject `mlock` syscall to lock the `mmap` allocated memory.

5 Evaluation

We have developed a proof-of-concept prototype of HyperShell with 3,700 lines of C code. The implementation is scattered across both the host OS side, which is atop Linux kernel 3.0.0-31, and the KVM side. While we have used KVM to build HyperShell, we believe our

Process	**S**	$B(ms)$	$D(ms)$	$T(X)$	date	✗	0.11	0.12	1.09	mkdir	✓	0.10	0.19	1.90
ps	✗	1.33	5.42	4.08	w	✗	0.95	6.62	6.97	mkfifo	✓	0.10	0.19	1.90
pidstat	✗	1.95	7.56	3.88	hostname	✓	0.04	0.06	1.50	mknod	✓	0.10	0.19	1.90
nice	✓	0.07	0.11	1.57	groups	✓	0.21	0.62	2.95	mv	✓	0.15	0.31	2.07
getpid	✓	0.01	0.02	2.00	hostid	✓	0.16	0.56	3.50	rm	✓	0.08	0.15	1.88
mpstat	✗	0.29	0.66	2.28	locale	✓	0.09	0.17	1.89	od	✓	0.12	0.35	2.92
pstree	✗	0.69	6.03	8.74	getconf	✓	0.09	0.34	3.78	cat	✓	0.07	0.18	2.57
chrt	✓	0.11	0.16	1.45	**System Utils**	**S**	$B(ms)$	$D(ms)$	$T(X)$	link	✓	0.07	0.13	1.86
renice	✓	0.11	0.18	1.64	uptime	✗	0.07	0.47	6.71	comm	✓	0.08	0.22	2.75
top	✗	504.92	510.85	1.01	sysctl	✓	8.5	42.72	5.03	shred	✗	0.72	0.92	1.28
nproc	✓	0.07	0.26	3.71	arch	✓	0.07	0.11	1.57	truncate	✓	0.07	0.26	3.71
sleep	✓	1.27	1.28	1.01	dmesg	✓	0.38	0.51	1.34	head	✓	0.07	0.15	2.14
pgrep	✓	0.89	4.72	5.30	lscpu	✓	0.26	1.21	4.65	vdir	✓	0.63	3.95	6.27
pkill	✓	0.87	4.33	4.98	mcookie	✗	0.29	0.49	1.69	nl	✓	0.08	0.17	2.13
snice	✓	0.17	0.65	3.82	**Disk/Devices**	**S**	$B(ms)$	$D(ms)$	$T(X)$	tail	✓	0.08	0.20	2.50
echo	✓	0.07	0.09	1.29	blkid	✓	0.14	0.61	4.36	namei	✓	0.07	0.13	1.86
pwdx	✓	0.05	0.07	1.40	badblocks	✓	0.35	0.44	1.26	whereis	✓	2.05	4.86	2.37
pmap	✓	0.16	0.36	2.25	lspci	✓	31.40	36.52	1.16	stat	✓	0.27	0.78	2.89
kill	✓	0.01	0.04	4.00	iostat	✓	0.45	1.04	2.31	readlink	✓	0.07	0.12	1.71
killall	✓	0.62	3.03	4.89	du	✓	0.11	0.53	4.82	unlink	✓	0.07	0.13	1.86
Memory	**S**	$B(ms)$	$D(ms)$	$T(X)$	df	✓	0.16	0.35	2.19	cut	✓	0.08	0.17	2.13
free	✗	0.04	0.08	2.00	**Filesystem**	**S**	$B(ms)$	$D(ms)$	$T(X)$	dir	✓	0.07	0.20	2.86
vmstat	✗	0.19	0.33	1.74	sync	✓	8.07	6.53	0.81	mktemp	✓	0.09	0.18	2.00
slabtop	✗	0.22	0.36	1.64	getcap	✓	0.04	0.08	2.00	rmdir	✓	0.07	0.13	1.86
Modules	**S**	$B(ms)$	$D(ms)$	$T(X)$	lsof	✓	3.31	6.12	1.85	ptx	✓	0.12	0.45	3.75
rmmod	✓	0.51	3.14	6.16	pwd	✓	0.07	0.11	1.57	chcon	✓	0.06	0.12	2.00
modinfo	✓	0.48	1.54	3.21	**Files**	**S**	$B(ms)$	$D(ms)$	$T(X)$	**Network**	**S**	$B(ms)$	$D(ms)$	$T(X)$
lsmod	✓	0.10	0.17	1.70	chgrp	✓	0.19	0.47	2.47	ifconfig	✗	0.32	1.15	3.59
Environment	**S**	$B(ms)$	$D(ms)$	$T(X)$	chmod	✓	0.07	0.14	2.00	ip	✓	0.10	0.20	2.00
who	✓	0.14	0.72	5.14	chown	✓	0.19	0.47	2.47	route	✓	138.65	150.32	1.08
env	✓	0.07	0.11	1.57	cp	✓	0.11	0.27	2.45	ipmaddr	✓	0.13	0.34	2.62
printenv	✓	0.07	0.1	1.43	uniq	✓	0.09	0.35	3.89	iptunnel	✓	0.09	0.29	3.22
whoami	✓	0.19	0.45	2.37	file	✓	0.87	1.72	1.98	nameif	✓	0.10	0.21	2.10
stty	✓	0.11	0.46	4.18	find	✓	0.20	0.58	2.90	netstat	✗	0.25	0.37	1.48
users	✓	0.09	0.53	5.89	grep	✓	0.35	2.14	6.11	arp	✓	0.14	0.24	1.71
uname	✓	0.09	0.11	1.22	ln	✓	0.08	0.14	1.75	ping	✗	15.02	18.2	1.21
id	✓	0.26	0.85	3.27	ls	✓	0.14	0.27	1.93	Avg.	-	7.27	8.45	2.73

Table 2: Evaluation Result of the Tested Utility Software. *S* stands for whether there is any Syntax-difference, $B(ms)$ stands for the average time of the base execution, $D(ms)$ stands for the average execution time of the utility in HyperShell when using the daemon mode in GVM, and $T(X)$ stands for the result of D/B (i.e., the times).

design can be applied to other types of hypervisors such as Vmware, Xen and VirtualBox.

In this section, we present our evaluation results. All of our experiments were carried out on a host machine configured with an Intel Core i7 CPU with 8G memory and running with Ubuntu 12.04 using Linux kernel 3.0.0-31; the guest OS is Debian 6.04 with kernel 2.6.32.8.

5.1 Effectiveness

Benchmark Software. Recall the goal of HyperShell is to enable the execution of native management utilities at the hypervisor layer to manage a guest OS, and also enable the fast development of these software by using the R-syscall abstraction. Since the software development with HyperShell is very simple (a hypervisor programmer just needs to annotate the syscall and inform HyperShell which one is an R-syscall), we skip this evaluation. In the following, we describe how we automatically execute the native utilities in HyperShell to transparently manage a guest OS.

Today, there are a large number of administrative utilities to manage an OS. To test HyperShell, we systematically examined all of the utilities (in total 198) from six packages including core-utility, util-linux, procps, module-init-tools, sysstat, and net-tools, and eventually we selected 101 utilities, as presented in Table 2, though technically we can execute all of them. The selection criteria is the following: if a utility is all user level program (e.g., hash computation such as `md5sum`), or not so system management related (e.g., `tr`), or can be executed in alternative way (e.g., `poweroff`, `halt`), or not supported by the kernel any more (e.g., `rarp`), we ignore them.

Experimental Result. Without any surprise, through our automated system call reverse execution policy, all of these utilities can be successfully executed in HyperShell. To verify the correctness of these utilities, we use a cross-view comparison approach in a similar way when we tested our prior systems such as Vmst [16, 17] and Exterior [18]. Basically, to test a given utility such as `ps`, we first execute it inside the GVM and save the output, which is called the in-VM view; then we execute it inside HyperShell to manage the GVM and also save the output, which is called the out-of-VM view. Then we compare the syntax (through `diff`) and semantics (with a manual verification) of the in-VM and out-of-VM views, which leads to the two sets of effectiveness test results: one is the syntax comparison, and the other is the semantic (i.e, the meaning) comparison.

We notice that while there are 16 utilities that have syntax differences (as shown in the *S* column in Table 2), all other utilities have the same screen output. A further investigation shows that the syntax differences among them is actually caused due to the different location (host OS vs. GVM) and timing of performing our in-VM and out-of-VM experiment. Regarding the semantics, we notice that all of the utilities have *the same semantics* as the original in-VM programs through our manual verification.

Testing w/ More Guest Kernels. Working at syscall level allows HYPERSHELL with less constraint and wider applicability because of the POSIX compatibility. For instance, we can now use a single host OS to manage a large number of syscall-compatible OSes. To validate this, we selected five other recently released Linux kernels of versions 2.6.32, 2.6.38, 3.0.10, 3.2.0, and 3.4.0, and executed them in our GVM. Our benchmark utilities were all correctly executed with these kernels.

5.2 Performance Overhead

When executing a program in HYPERSHELL, there are two processes to fulfill the execution: the master process executed in the host OS, and the helper process executed in the guest OS. Consequently, we have to measure two sets of performance. One is how slow an end-user would feel when executing a utility in HYPERSHELL. The other is the impact with respect to the guest OS kernel due to our syscall capturing and helper process execution at the GVM. Below we report these two types of overhead.

5.2.1 Performance Impact to the Native Utilities

With different settings of the helper process (daemon or non-daemon), we could also have two sets of performance overhead for the utility software. However, the performance differences for these two settings mainly come from the creation of the helper process, which is almost a constant factor (the time interval between the two scheduled executions of the `init` process). Our evaluation shows that every 5 seconds, the `init` process will be scheduled. Therefore, it leads to the creation of a helper process with maximum 5 seconds, the worst case delay if we want to use a non-daemon helper process to execute the R-syscall. All other latency is the same compared to the daemon mode execution. Therefore, in the following, we present our result with the daemon mode execution of the helper process.

Again, we used these 101 utilities in effectiveness tests to measure this overhead. Specifically, we executed the utilities each with 100 times and computed their average. First, we ran all of them in a native-KVM and got the average execution time for each of them as the base. This result is presented in the *B*-column of Table 2. Then we collected the average run time of these utilities in HYPERSHELL with a daemon helper process in the GVM. This result is presented in the *D*-column. We computed the overhead of this test with the base one, and we report them in the *T*-column. We compare with the execution running in native-KVM instead of native host OS because we are comparing our out-of-VM approach with an in-VM approach. We notice that on average, with a daemon mode helper process, HYPERSHELL has 2.73X slowdown compared to the executions running in a native-KVM. This overhead mainly comes from the data exchange and synchronization between the host OS and the GVM during the R-syscall execution.

Tested Item	Native-KVM	GVM-RI-Phase	Slowdown (%)	GVM-RE-Phase	Slowdown (%)
stat (μs)	0.39	2.28	82.89	0.41	4.88
fork proc (μs)	47.20	147.26	67.95	47.54	0.72
exec proc (μs)	158.20	480.00	67.04	161.30	1.92
sh proc (μs)	384.90	1088.10	64.63	386.30	0.36
ctxsw (μs)	0.59	1.23	52.03	0.73	19.18
10K File Create (μs)	17.80	40.67	56.23	17.96	0.89
10K File Delete (μs)	4.64	7.16	35.20	4.65	0.22
Bcopy (MB/s)	5689.17	5647.71	0.73	5605.40	1.47
Rand mem (ns)	72.20	72.65	0.62	73.24	1.42
Mem read (MB/s)	10150.00	10000.00	1.48	10000.00	1.48
Mem write (MB/s)	8567.70	8543.00	0.29	8540.40	0.32

Table 3: Micro-benchmark Test Result of GVM.

5.2.2 Performance Impact to the GVM

The performance impact to the GVM also falls into two scenarios: one is the system wide sysenter/sysexit interception that is used to capture the `init` process (recall we name it the RI-Phase), and the other is the R-syscall execution that occurs in the helper process (we call this RE-Phase). These two phases inevitably introduce performance penalty to the running workloads/processes at GVM. Note that if the GVM is neither running in RI nor RE-Phase, there is no performance overhead. To quantify the overhead from these two scenarios, we used standard benchmark programs (e.g., LMBench, and ApacheBench) that are used in other work (e.g., [37, 39]) to measure the runtime overhead of the guest OS execution at both micro and macro level for these two phases.

Also, according to the result from Table 2, the execution of the RE-phase is very short (on average 8.45 milliseconds). In addition, our RI will never be executed if the helper process has created. Therefore, we have to create an environment to keep executing RI and RE such that we can measure the impact to the long running benchmark programs. That is, we will keep polling the `init` process to measure the impact from the RI-Phase, and keep executing the `int3` loop for the helper process to measure the impact from the RE-Phase. These results are the worst case performance impact to running processes in the GVM.

Micro-benchmarks. To evaluate the primitive level performance slowdown, we used LMBench suites. In particular, we focused on the overhead of the `stat` syscall, process creation (fork proc), process execution (exec proc), C library function (sh proc), context switches (`ctxsw`), memory-related operations (e.g., bcopy, Mem read, Mem Write), and IO-related operations (e.g., 10k File Create, and 10K File Delete).

The detailed result is presented in Table 3. The RI-Phase tends to have large overhead on tests which contain syscalls, as we intercept the system-wide syscall entry and exit points. While we do not intercept context switches, our system still has large overhead on the `ctxsw` test. The reason is that LMBench tests the time of context switches on a number of processes. And these process are connected using pipe. Therefore, the measurement still contains syscalls. In contrast, during the RE-Phase, our syscall interception is only within the helper process, and it has significantly less overhead except the `ctxsw` case with similar reasons in the RI-Phase.

Macro Benchmarks. We used four real world workloads to quantify the performance slowdown at the macro level. In particular, we decompressed a source tarball of Linux 2.6.32.8 using bzip, and then compiled the kernel using kbuild. We recorded the process time. In the test of Apache, we used ApacheBench [1] to issue 100,000 requests for a 4k-byte file from a client machine and got the throughput (#request/s). For memcached [5], we recorded the time of processing 1,000 requests.

The performance overhead is presented in Table 4. For the RI-Phase, the overhead comes from the `VMexit` of trapping syscall entry and exit. Hence, the workloads that have large portions of IO operation will incur large overhead, e.g., as in Kbuild, Apache, and memcached. The worst case is memcached which is also sensitive to IO-latency. In contrast, computation intensive workloads have small overhead (as in the bzip case). Regarding the RE-Phase, all the workloads have small overhead because our system only introduces a user mode `int3` loop. The `VMexit` only occurs in the helper process execution context. The only side effect is that the helper process takes some CPU time slices from them.

5.3 Case Studies

Once we have enabled the execution of native utilities in HyperShell to manage the guest OS, many new use cases would appear. For instance, we can now kill malicious processes, remove malicious drivers, change the guest IP address, update the firewall rules, etc., directly from the hypervisor layer. In the following, we demonstrate an interesting use case of our system—full disk encryption (FDE) protected virus scanning from the hypervisor.

Today, because of the privacy and data-breach concerns, a growing practice for outsourced VMs is to deploy FDE. Unfortunately, this has brought challenges for disk introspection, forensics, and management. With HyperShell, we can actually use off-the-shelf anti-virus software from the host OS to transparently scan files in the guest OS even though the GVM disk might have been encrypted by FDE.

Benchmark Program	Native-KVM	GVM-RI-Phase	Slowdown (%)	GVM-RE-Phase	Slowdown (%)
bzip (s)	16.83	18.35	8.28	17.04	1.23
kbuild (s)	1799.00	2270.25	20.76	1889.97	4.81
memcached (s)	1.57	3.11	49.52	1.64	4.27
Apache (#request/s)	1104.60	904.12	18.15	1065.28	3.56

Table 4: Macro-benchmark Test Result of GVM.

To validate this, we installed `dm-crypt` [3], a transparent FDE subsystem in the Linux kernel (since version 2.6) in our GVM. Under a `test` user home directory, we copied a large volume of files including the source code of Linux-2.6.32.8, gcc, glibc, QEMU, Apache, and Lmbench, as well as two viruses from `offensivecomputing.com`, resulting in a total number of 101,415 files adding up to 1336.09 megabytes in size. In the host OS, we installed ClamAV-0.98 [2] and used it (in particular its `clamscan`) to scan the files in `/home/test` in the GVM. We tried two different approaches in this testing:

- The first is to directly allow `clamscan` running in HyperShell to scan the files in the GVM by redirecting the R-syscall, and in this case it took 188.35 seconds to scan the entire 1336.09 megabytes of files and find the two viruses.
- The second is to copy (i.e., `cp`) the files in `/home/test` to our host OS, and then scan them natively. In this case, it took 59 seconds to copy these files, with another 120.91 seconds scanning them, resulting in a total of 179.91 seconds.

It is worth noting that very interestingly if we installed ClamAV inside the GVM and scanned these files, it would take 271.58 seconds. Therefore, by moving certain management software running into HyperShell, it can in fact speedup certain computation (188.35 vs. 271.58) as shown in our `clamscan` case. There are two primary sources for this speedup: one is that there is no additional VMexit when processing the disk IO at the host OS side (i.e., IO in host OS is usually faster than guest OS), and the other is because there is no need for the decryption of the signature data base of ClamAV when running at host OS.

6 Limitations and Future Work

While HyperShell offers better automation (e.g., no need of login), uniformity (e.g., all of the VM can be checked for anti-virus), and centralized management (e.g., using only one copy of the software running at a hypervisor to manage a large number VMs, and there is a need of only updating the copy at the hypervisor layer), it comes with price. In particular, it will circumvent all of the existing user login and system audit for each managed VM.

For instance, syslog in each individual VM will not be able to capture all the executed events inside the guest OS. To fix this, we need to add a new log record at the hypervisor layer for each activity executed in HYPERSHELL, such that the entire cloud can still be audited. One avenue of our future research will address this.

Second, as normal utility software does, HYPERSHELL requires the trust of the guest OS kernel as well as the init process. Consequently, it cannot be used for security critical applications, especially when the kernel has been compromised. Also, unlike introspection, which aims to achieve stealthiness, HYPERSHELL is not designed with this goal in mind, since its primary goal is to manage the guest OS (which definitely introduces footprints) from out-of-VM in the same way as we manage in-VM but in a more centralized and automated manner.

Third, our current prototype requires both OSes running in the host OS and the GVM to have compatible syscall interface. If a guest OS uses a randomized system call interface (e.g., RandSys [25]), it could thwart the execution of the management utilities at HYPERSHELL. In fact, we can design certain logic in our Syscall Dispatcher and Reverse Syscall Execution component to perform syscall translations even though the syscalls are not fully compatible or randomized (e.g., with different syscall number). We leave this as another future work. Again, we would like to emphasize that working at syscall boundary makes HYPERSHELL with less constraint when compared to other alternative approaches. For instance, it is possible to directly inject the shell command to the guest OS to achieve the same goal (e.g., configure the guest OS), or directly inject the file system updates. However, command-line interfaces or configuration file interfaces are less stable when compared to the syscall interface. That is why eventually it leads to our R-syscall based approach.

Finally, our Syscall Dispatcher uses dynamic library interposition, and it ignores the syscall policy checking in the dynamic loader. Therefore, static linked native utilities cannot be executed in HYPERSHELL. Also, if there is a different loader whose syscall can be captured by library interposition, we have to design new techniques to differentiate the syscall policy for these syscalls. One possible solution is to add the call stack context in our policy check. In addition, while most of our design is OS-agnostic, we currently only demonstrate HYPERSHELL with the Linux kernel and we would like to test with other OSes such as Microsoft Windows. We leave these in our other future efforts.

7 Related Work

Our work is related to the virtual machine introspection (VMI) [19, 26] and VM management in the cloud. In this section, we review and compare them with HYPERSHELL.

Being a layer below of the OS, virtual machine gives new opportunities for VMI, which inspects and analyzes both the user level program and OS kernel states outside the machine itself. However, the key challenge in VMI lies in how to bridge the semantic gap. Over the past decade, many approaches have been proposed to address this problem, and these approaches can be classified into: debugger assisted (e.g., [19]), manual kernel data structure traversal (e.g., [24]), kernel source code analysis and customization (e.g., [8, 23]), in-VM kernel module assisted (e.g., [30, 34]), and binary code or execution context reuse (e.g., [21, 15, 35, 22, 16, 17, 18]). In this section, we will not go through and compare with each of these existing techniques, but rather compare with the most related ones as presented in Table 5.

To narrow the semantic gap, VIRTUOSO [15] made a first step showing that we can actually reuse the legacy binary code to automatically create VMI tools with the assistance from a human expert. The key idea is to first train each in-VM program (e.g., ps) and then translate the trained traces (essentially slices) into an independent introspection program running at the hypervisor layer. Inspired by VIRTUOSO, VMST [16] shows a dual-VM based, online kernel data redirection approach that addresses the limitations from the training (i.e., code coverage issue). Its key idea is to reuse the execution context of an inspection process in an SVM; when a kernel instruction accesses the kernel data of introspection interest, it redirects the data from the GVM to the SVM. Built atop VMST, EXTERIOR [18] demonstrates that it is feasible to build an external shell to perform the out-of-VM guest OS writable operations (e.g., for configuration) [28]. Similar to VIRTUOSO, both VMST and EXTERIOR do not need to trust the guest-OS kernel.

While EXTERIOR [18] has made an early attempt of building a hypervisor layer shell, it has a lot of constraints and is far from practical. Specifically, it has to first perform the guest OS fingerprinting [20], and then use the exact same version of the guest OS running in an SVM to introspect the kernel state of a GVM. Second, it can suffer from various failures and shortfalls when an introspection related syscall uses kernel synchronization primitives [16, 18]. Third, it is built atop a binary code translation based VM (e.g., QEMU [6]), which often has 10X-40X performance slowdown compared to the native execution (though recently HybridBridge [32] has improved the performance with one order of magnitude). Finally, it is mainly for introspection and has very limited functionality (e.g., it ignores the disk data including the swapped memory).

Process Implanting (PI) [21] shows that we can inject a process running into a GVM by reusing an existing process context. At a high level, HYPERSHELL does share some similarity regarding the process injection. However, PI has only limited functionality. For instance, it cannot directly copy a file from inside to outside. It also cannot observe the output from native software, unless rewriting the utility with hypercall (a para-virtualization approach that is not transparent to the guest OS). In addition, it requires the recompilation of the injected programs with static linking; in contrast, HyperShell is transparent to both utility software and the guest OS.

Designed for process monitoring, process out-grafting (Pog) [35] relocates a suspect process from a GVM to an SVM, and then uses a trusted security tool (e.g., `strace`) in the SVM to monitor the behavior of the suspect process. Unlike HyperShell, which selectively redirects the syscall based on a transparent policy, all the syscalls of the suspect process are redirected from the SVM to the GVM. Therefore, Pog does not face the challenges as in HyperShell to differentiate the syscall redirection policy. Meanwhile, all the applications supported by Pog can certainly be supported by HyperShell, but not vice versa.

Designed for high performance computing, Gears [22] shows that we can push certain VMM level virtual services for a guest into the guest itself. Through such a way, we can reduce the implementation complexity (since there is no semantic gap for in-VM programs) and increase the performance. At a high level, while Gears and HyperShell shares some similarity of using syscall interception and code injection techniques, the substantial difference is that Gears is not a binary code reuse based approach, and it is not transparent to the in-VM programs and requires programmer's efforts to (re)develop the new software.

Most recently, concurrent to HyperShell, ShadowExecution [38] also explores the concept of system call redirection and process injection. With a number of other security means such as process image protection (e.g., code and data integrity) and runtime execution protection (e.g., control flow integrity) of both guest OS kernel and the injected process, ShadowExecution shows that VMI tools can be built as in VMST. The difference compared to ShadowExecution is that HyperShell is mainly designed for Cloud in-VM management, whereas ShadowExecution is mainly for security, though they both are based on the system call redirection concept.

Our work is also related to VM management in the cloud, such as VM cloning (e.g., [27]), VM migration (e.g., [10]), and VM replication (e.g., [12]). However, these management techniques treat each VM as a whole. In contrast, we aim to design programs to manage each guest OS at a fine grained level from our HyperShell, much like the way we manage an OS in-VM.

8 Conclusion

We have presented the design, implementation, and evaluation of HyperShell, a practical hypervisor layer shell for automated, uniformed, and centralized guest OS management. To overcome the semantic gap challenge, we introduce a reverse system call abstraction, and we show that this abstraction can be transparently implemented. Resulting from this, many of the legacy guest OS management utilities can be directly executed in HyperShell. Our empirical evaluation with 101 native Linux utilities shows that we can use HyperShell to manage a guest OS directly from the hypervisor layer without requiring any access to administrator's account. Regarding the performance, it has on average 2.73X slowdown for the tested utilities compared to their native in-VM execution, and less than 5% overhead to the guest OS kernel.

Systems	Execution Context Reuse	wo/ Dual-VM Architecture	wo/ Identical Kernel	wo/ Trust to Guest Kernel	High Code Coverage	Fully Automated	Memory Introspection	Disk Introspection	Guest Management	Process Monitoring
Virtuoso	✗	✓	✗	✓	✗	✗	✓	✗	✗	✗
Vmst	✓	✗	✗	✓	✓	✓	✓	✗	✗	✗
Exterior	✓	✗	✗	✓	✓	✓	✓	✗	✓	✗
Pi	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗
Pog	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓
Gears	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
HyperShell	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓

Table 5: Comparison with the most related work.

Acknowledgment

We are grateful to Jon Howell and other anonymous reviewers for their insightful comments. We also thank Erick Bauman and Kenneth Miller for their valuable feedback on an early draft of this paper. This research was supported in part by a research gift from VMware Inc. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the VMware.

References

[1] Apachebench - apache http server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[2] Clamav, an open source (gpl) antivirus engine designed for detecting trojans, viruses, malware and other malicious threats. http://www.clamav.net/.

[3] dm-crypt, linux kernel device-mapper crypto target. http://code.google.com/p/cryptsetup/wiki/DMCrypt.

[4] Kernel based virtual machine. *http://www.linux-kvm.org*.

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

[6] QEMU: an open source processor emulator. *http://www.qemu.org/*.

[7] Skytap announces over one million virtual machines launched. http://www.skytap.com/news-events/press-releases/skytap-announces-over-one-million-virtual-machines-launched, January 2012.

[8] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X. Mapping kernel objects to enable systematic integrity checking. In *The 16th ACM Conference on Computer and Communications Security (CCS'09)* (Chicago, IL, USA, 2009).

[9] Chen, P. M., and Noble, B. D. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS'01)* (Napa, CA, USA, 2011).

[10] 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 conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)* (Boston, MA, USA, 2005).

[11] CLARK, D. D. The structuring of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles (SOSP'85)* (Orcas Island, Washington, USA, 1985).

[12] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)* (Berkeley, CA, USA, 2008).

[13] CURRY, T. W. Profiling and tracing dynamic library usage via interposition. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1 (ATC'94)* (Boston, Massachusetts, USA, 1994).

[14] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security (CCS'08)* (Alexandria, Virginia, USA, 2008).

[15] DOLAN-GAVITT, B., LEEK, T., ZHIVICH, M., GIFFIN, J., AND LEE, W. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the* 32^{nd} *IEEE Symposium on Security and Privacy (S&P'11)* (Oakland, CA, USA, 2011).

[16] FU, Y., AND LIN, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *Proceedings of* 33^{rd} *IEEE Symposium on Security and Privacy (S&P'12)* (San Francisco, CA, USA, 2012).

[17] FU, Y., AND LIN, Z. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Trans. Inf. Syst. Secur. 16*, 2 (2013).

[18] FU, Y., AND LIN, Z. Exterior: Using a dual-vm based external shell for guest-os introspection, configuration, and recovery. In *Proceedings of the Ninth Annual International Conference on Virtual Execution Environments (VEE'13)* (Houston, TX, USA, 2013).

[19] GARFINKEL, T., AND ROSENBLUM, M. A virtual machine introspection based architecture for intrusion detection. In *Proceedings Network and Distributed Systems Security Symposium (NDSS'03)* (San Diego, CA, USA, 2003).

[20] GU, Y., FU, Y., PRAKASH, A., LIN, Z., AND YIN, H. Ossommelier: Memory-only operating system fingerprinting in the cloud. In *Proceedings of the 3rd ACM Symposium on Cloud Computing (SOCC'12)* (San Jose, CA, USA, 2012).

[21] GU, Z., DENG, Z., XU, D., AND JIANG, X. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the 30th IEEE Symposium on Reliable Distributed Systems (SRDS'11)* (Madrid, Spain, 2011).

[22] HALE, K. C., XIA, L., AND DINDA, P. A. Shifting gears to enable guest-context virtual services. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC'12)* (San Jose, California, USA, 2012).

[23] HOFMANN, O. S., DUNN, A. M., KIM, S., ROY, I., AND WITCHEL, E. Ensuring operating system kernel integrity with osck. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS'11)* (Newport Beach, California, USA, 2011).

[24] JIANG, X., WANG, X., AND XU, D. Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)* (Alexandria, Virginia, USA, 2007).

[25] JIANG, X., WANGZ, H. J., XU, D., AND WANG, Y.-M. Randsys: Thwarting code injection attacks with system service interface randomization. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)* (Beijing, China, 2007).

[26] 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 (SOSP'05)* (Brighton, United Kingdom, 2005).

[27] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)* (Nuremberg, Germany, 2009).

[28] LIN, Z. Toward guest os writable virtual machine introspection. *VMware Technical Journal 2*, 2 (2013).

[29] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'05)* (San Diego, CA, USA, 2005).

[30] PAYNE, B. D., CARBONE, M., SHARIF, M. I., AND LEE, W. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of 2008 IEEE Symposium on Security and Privacy (S&P'08)* (Oakland, CA, May 2008).

[31] PFOH, J., SCHNEIDER, C., AND ECKERT, C. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, vol. 7038 of *Lecture Notes in Computer Science*. Springer, Nov. 2011, pp. 96–112.

[32] SABERI, A., FU, Y., AND LIN, Z. Hybrid-bridge: Efficiently bridging the semantic-gap in virtual machine introspection via decoupled execution and training memoization. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)* (San Diego, CA, USA, 2014).

[33] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A Tiny Hypervisor to Guarantee Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'07)* (Stevenson, WA, USA, 2007).

[34] SHARIF, M. I., LEE, W., CUI, W., AND LANZI, A. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS'09)* (Chicago, Illinois, USA, 2009).

[35] SRINIVASAN, D., WANG, Z., JIANG, X., AND XU, D. Process out-grafting: an efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS'11)* (Chicago, Illinois, USA, 2011).

[36] TSAI, T. K., AND SINGH, N. Libsafe: Transparent system-wide protection against buffer overflow attacks. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)* (Washington, DC, USA, 2002).

[37] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy (S&P'10)* (Okaland, CA, USA, 2010).

[38] WU, R., CHEN, P., LIU, P., AND MAO, B. System call redirection: A practical approach to meeting real-world vmi needs. In *Proceedings of the 44th International Conference on Dependable Systems and Networks (DSN'14)* (Atlanta, Georgia, June 2014).

[39] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Cascais, Portugal, 2011).

XvMotion: Unified Virtual Machine Migration over Long Distance

Ali José Mashtizadeh[⋆]*, Min Cai, Gabriel Tarasuk-Levin*
Ricardo Koller, Tal Garfinkel[†]*, Sreekanth Setty*
Stanford University[⋆] *– VMware, Inc. – Unaffiliated*[†]

Abstract

Live virtual machine migration allows the movement of a running VM from one physical host to another with negligible disruption in service. This enables many compelling features including zero downtime hardware upgrades, dynamic resource management, and test to production service migration.

Historically, live migration worked only between machines that shared a common local subnet and storage system. As network speed and flexibility has increased and virtualization has become more pervasive, wide area migration is increasingly viable and compelling. Ad-hoc solutions for wide area migration have been built, combining existing mechanisms for memory migration with techniques for sharing storage including network file systems, proprietary storage array replication or software replicated block devices. Unfortunately, these solutions are complex, inflexible, unreliable and perform poorly compared to local migration, thus are rarely deployed.

We have built and deployed a live migration system called XvMotion that overcomes these limitations. XvMotion integrates support for memory and storage migration over the local and wide area. It is robust to the variable storage and network performance encountered when migrating long distances across heterogeneous systems, while yielding reliability, migration times and downtimes similar to local migration. Our system has been in active use by customers for over a year within metro area networks.

1 Introduction

Live virtual machine (VM) migration moves a running VM between two physical hosts with as much transparency as possible: negligible downtime, minimal impact on workload, and no disruption of network connectivity.

Originally, VM migration only moved a VM's memory and device state between two closely related physical hosts within a cluster i.e., hosts that shared a common storage device, usually a storage array accessed via a dedicated SAN, and a common Ethernet subnet to enable network mobility. Assumptions about host locality made sense given the limited scale of VM adoption, and limitations of storage and network devices of the past.

Today, many of these assumptions no longer hold. Data center networks are much faster with 10 Gbps Ethernet being common, and 40/100 Gbps adoption on the way. Tunneling and network virtualization technologies [14, 20, 22] are alleviating network mobility limitations. Different workloads and high performance SSDs have made local shared-nothing storage architectures increasingly common. Finally, VM deployments at the scale of tens of thousands of physical hosts across multiple sites are being seen. Consequently, the ability to migrate across clusters and data centers is increasingly viable and compelling.

Some have attempted to build wide area live migration by combining techniques to migrate storage e.g., copying across additional storage elements, proprietary storage array capabilities, or software storage replication in conjunction with existing technologies for migrating memory and device state. These ad-hoc solutions are often complex and fragile—incurring substantial penalties in terms of performance, reliability, and configuration complexity compared with doing migration locally. Consequently, these systems are deployed at a very limited scale and only by the most adventuresome users.

We present XvMotion, an integrated memory and storage migration system that does end-to-end migration between two physical hosts over the local or wide area. XvMotion is simpler, more resilient to variations in network and storage performance, and more robust against failures than current ad-hoc approaches combining different storage replication and memory migration solutions, while offering performance, i.e., workload impact and service downtimes, comparable to that of a local migration.

XvMotion has been in use with customers for over a year, and seen wide spread application at the data center and metro-area scale, with link latencies in the tens of milliseconds. Our results with moving VMs between Palo Alto, California and Banglore, India demonstrate that migrations on links with latencies in the hundreds of milliseconds are practical.

We begin by examining how current local migration architectures developed and survey ad-hoc approaches used to support wide area migration. Next, we explore our design and implementation of XvMotion: our bulk transport layer, our approach to asynchronous storage mirroring, our integration of memory and storage movement, our workload throttling mechanism to control page dirtying rates—allowing migration across lower bandwidth links, and our disk buffer congestion control mechanism to support migration across hosts with heterogeneous storage performance. We then present optimizations to minimize switch over time between the old (pre-migration) and new (post-migration) VM instance, cope with high latency and virtualized networks, and reduce virtual disk copy overhead. Our evaluation explores our local and long distance migration performance, and contrasts it with existing local live storage migration technologies on shared storage. After this, we discuss preventing split-brain situations, network virtualization, and security. We close with related work and conclusions.

2 The Path to Wide Area Migration

Wide area live migration enables a variety of compelling new use cases including whole data center upgrades, cluster level failure recovery (e.g. a hardware failure occurs, VM's are migrated to a secondary cluster and then migrated back when the failure has been fixed), government mandated disaster preparedness testing, disaster avoidance, large scale distributed resource management, and test to production data center migration. It also allows traditional mainstays of VM migration, like seamless hardware upgrades and load balancing, to be applied in data centers with shared-nothing storage architectures.

Unfortunately, deploying live migration beyond a single cluster today is complex, tedious and fragile. To appreciate why current systems suffer these limitations, we begin by exploring why live migration has historically been limited to a single cluster with a common subnet and shared storage, and how others have tried to generalize live migration to the wide area. This sets the stage for our discussion of XvMotion in the next section.

Live memory and device state migration, introduced in 2003, assumed the presence of shared storage to provide a VM access to its disks independent of what physical host it ran on and a shared subnet to provide network mobility. In 2007, live storage migration was introduced, allowing live migration of virtual disks from one storage volume to another assuming that both volumes were accessible on a given host. This provided support for storage upgrades and storage resource management. However, the ability to move a VM between two arbitrary hosts without shared storage has largely been limited to a few research systems [9,28] or through ad-hoc solutions discussed later in this section.

Why not migrate an entire VM, including memory and storage, between two arbitrary hosts over the network? Historically, hardware limitations and usage patterns explain why. Data centers of the previous decades used a mix of 100 Mbps and gigabit Ethernet, with lower cross sectional bandwidths in switches than today's data centers. High performance SAN based storage was already needed to meet the voracious IO demands induced by consolidating multiple heavy enterprise workloads on a single physical host. Network mobility was difficult and complex, necessitating the use of a single shared subnet for VM mobility. Finally, common customer installations were of modest size, and live migration was used primarily for hardware upgrades and load balancing, where limiting mobility to a collection of hosts sharing storage was an acceptable constraint.

Many of these historical limitations no longer apply. Data center networks are much faster with 10 Gbps Ethernet being common, and 40/100 Gbps adoption on the way, with a correspond growth in switch capacity. The introduction of tunneling and network virtualization technologies, like Cisco OTV [14], VXLAN [20] and OpenFlow [22], are alleviating the networking limitations that previously prevented moving VMs across Ethernet segments. With changes in workloads and the increased performance afforded by SSDs, the use of local shared-nothing storage is increasingly common. Finally, with users deploying larger scale installations of thousands or tens of thousands of physical hosts across multiple sites, the capacity to migrate across clusters and data centers becomes increasingly compelling.

To operate in both the local and wide area, two primary challenges must be addressed. First, existing live migration mechanisms for memory and device state must be adapted to work in the wide area. To illustrate why, consider our experience adapting this mechanism in ESX.

In local area networks, throughputs of 10 Gbps are common, as our migration system evolved to support heavier and larger workloads we designed optimizations and tuned the system to fit local area networks. In contrast, metro area links are in the range of a few gigabits at most, and as distances increase in the wide area throughputs typically drop below 1 Gbps with substantial increases in latency. On initial runs on the wide area our local live migration system didn't fare well; we often saw downtimes of 20 seconds or more causing service interrupting failures in many workloads, migrations were frequently unable to complete, and network bandwidth was underutilized. To understand why, lets briefly recap how live memory and device state migration work.

Production live migration architectures for memory and device state all follow a similar iterative copy approach first described by Nelson *et al.* [24]. We initially mark all pages of the VM as dirty, and begin to iterate through memory, copying pages from source to destination. After a page is copied, we install a write trap and mark the page as clean. If the write trap is triggered, we again mark the page as dirty. We apply successive "iterative pre-copy" passes, each pass copying remaining dirty pages. When the remaining set of pages to be copied is small enough to be copied without excessive downtime (called convergence), we suspend VM execution, and the remaining dirty pages are sent to the destination, along with the device state. Finally, we resume our now fully consistent VM on the destination and kill our source VM.

Unfortunately, naively applied, this approach does not behave well over slower and higher latency networks. Workloads can easily exceed the memory copy throughput, preventing the migration from every converging and causing large downtimes. To cope with this, we introduced workload throttling (§ 3.5) to limit the downtime.

Compounding that issue our initial local migration system used a single TCP connection as transport that suffers from head-of-line blocking and underutilizes network throughput. An improved transport mechanism(§ 3.2) and TCP tuning(§ 4.3) were required to fully utilize high bandwidth-delay links.

The other major challenge to long distance migration is how to move storage. Today, users with the desire to migrate VMs within or across data centers rely on three different classes of ad-hoc approaches: sharing storage using a network file system as a temporary "scratch" space for copying the VM, proprietary storage array based replication, and software based replication (e.g., DRBD) done outside the virtualization stack. Each approaches exhibits particular limitations, understanding these helps to motivate our approach.

In the first approach, a user exploits a network file system (e.g., NFS or iSCSI) to provide a temporary scratch space between two hosts, first doing a storage migration to this scratch space from the source host, then doing a second storage migration from the scratch space to the destination host. A live memory migration is done in between to move memory. This approach has several unfortunate caveats.

Moving data twice doubles total migration time—the performance impact is generally worse than this as the storage migration is not coordinated with memory and device state movement. Running the VM temporarily over the WAN while its disk is located on the scratch volume further penalizes the workload. Finally, hop-by-hop approaches are not atomic, if network connectivity fails, the VMs disk could end up on one side of the partition, and its memory on the other—this state is unrecoverable and the VM must be powered off.

Seeing the limitations of this approach, storage vendors stepped in with proprietary solutions for long distance storage migrations, such as EMC VPLEX [8]. These solutions use synchronous and asynchronous replication applied at a LUN level to replicate virtual disks across data centers. These solutions typically switch from asynchronous to synchronous replication during migration to ensure an up to date copy of the VM's storage is available in both data centers to eliminate the risk that a network partition will crash the VM, unfortunately, this imposes a substantial penalty on workloads during migration. These approaches are not cross-host migration as they do not support shared nothing storage configurations, instead these migrations are between two compatible storage arrays in different data centers.

The last approach relies on software replication, such as DRBD [2], to create a replicated storage volume. DRBD is a software replication solution, where typically each disk is backed by a DRBD virtual volume. To migrate one would back the VM by a DRBD volume, then replicate it to the desired destination machine. Next, a live migration would migrate the VM's memory/execution to the destination. Once complete the DRBD master is switched to the destination and the replication is terminated. A case study of a DRBD deployment [25] uses asynchronous replication during the bulk disk copy and synchronous storage replication for the duration of the migration, as noted above, this synchronous replication phase imposes an additional overhead on the VM workload. Another issue encountered with this approach is the lack of atomicity of the migration, i.e., the storage replication and memory migration system must simultaneously synchronize and transfer ownership from source to destination. Otherwise, this extends the window for a network partition causing the VM to power-off or be damaged.

3 XvMotion

XvMotion provides live migration between two arbitrary end hosts over the local or wide area. Many of limitations of previous approaches are eliminated by being a purely point-to-point solution, without the need for support from a storage system or intermediate nodes.

Our primary goals are to provide unified live migration (memory and storage), with the critical characteristics of local migration—atomic switch over, negligible downtime, and minimal impact on the VM workload, that can operate in the local and wide area.

We present XvMotion as follows. We begin with an overview of the XvMotion architecture in § 3.1. Next, we examine our bulk transport layer *Streams* in § 3.2. In § 3.3 we explore our implementation of asynchronous storage mirroring and storage deduplication. The coordination of memory and storage copying is described in § 3.4. § 3.5 presents SDPS, a throttling mechanism used to limit a VM's page dirtying rate to adapt to available network bandwidth and ensure that a migration converges. § 3.6 describes disk buffer congestion control, which allows XvMotion to adapt to performance differences between disks on the source and destination hosts.

3.1 Architecture Overview

XvMotion builds on the live migration [24] and IO Mirroring [21] mechanisms in ESX. Live memory migration is implemented using an iterative copy approach described in the previous section. ESX uses synchronous mirroring to migrate virtual disks from one volume to another on the same physical host. We augment this with asynchronous IO mirroring for migrating virtual disks across hosts while hiding latency.

Figure 1 depicts the XvMotion architecture. The *Streams* transport handles bulk data transfers between the two hosts. The *Live Migration* module is responsible for the many well-known tasks of VM live memory migration, such as locating VM memory pages for transmission, and appropriate handling of the VM's virtual device state. It enqueues its pages and relevant device state for transmission by *Streams*.

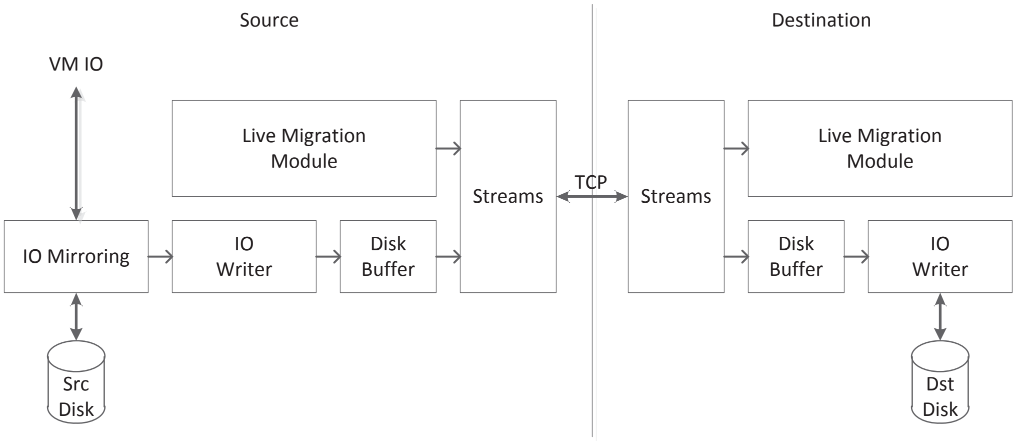


Figure 1: Architecture Overview: shows the main components involved in cross host migration for both the source and destination. IO Mirroring pushes data synchronously into the disk buffer through the migration IO Writer interface. The streams transport is responsible for bulk transfer of data between the hosts. Finally, on the destination the data is written to the disk.

The *IO Mirroring* module interposes on all virtual disk writes and mirrors these to the XvMotion *IO Writer* module. The *IO Mirroring* module is also responsible for the bulk reading of storage blocks during the initial copy i.e., synchronizing the mirror. The IO Writer module on the destination is responsible for dequeuing and writing blocks to the destination virtual disk.

The IO Writer then enqueues its data for the Disk Buffer. Disk buffering enables for asynchronous IO mirroring by storing a copy in RAM until it reaches the destination. While buffer space is available disk mirroring will acknowledge all mirror writes immediately, which eliminates the performance impact of network latency.

3.2 Streams Transport Framework

Streams is a bulk transport protocol we built on top of TCP for transferring memory pages and disk blocks. Streams creates multiple TCP connections between the hosts, including support for multipathing across NICs, IP addresses, and routes. Additional discussion of how Streams uses TCP is given in § 4.1 and § 4.3.

Using multiple connections mitigates head of line blocking issues that occur with TCP, leading to better utilization of lossy connections in the wide area [16, 17]. In the local area, the multiple independent connections are used to spread the workload across cores to saturate up to 40 Gbps links.

Streams dynamically load-balances outgoing buffers over the TCP connections, servicing each connection in round-robin order as long as the socket has free space. This allows us to saturate any number of network connections, up to PCI bus limitations. As a consequence of this approach, data is delivered out-of-order. However, there are no ordering requirements within an iteration of a memory copy: we just need to copy all pages once in any order. Any ordering requirement is expressed using a *write barrier*. For example, we require write barriers between memory pre-copy iterations.

Write barriers are implemented in the following way. Upon encountering a barrier request, the source host transmits a barrier message over all communication channels to the destination host. The destination host will read data on each channel up to the barrier message, then pause until all channels have reached the barrier message. Barriers are an abstraction provided to memory migration and storage migration, but do not necessarily prevent Streams from saturating the network while waiting for all connections to reach the barrier.

The general lack of ordering in Streams impacts our storage migration design. As we see in the next section, barriers are important as ordering is sometimes necessary for correctness.

3.3 Asynchronous IO Mirroring

Storage migration works by performing a single pass, bulk copy of the disk, called the *clone process*, from the source host to the destination host. Concurrently, IO mirroring reflects any additional changes to the source disk that occur during the clone process to the destination. When the clone process completes, the source and destination disks are identical [21]. Both clone and mirroring IOs are made asynchronously by using the Disk Buffer and Streams framework. This minimizes the performance impact on our source VM, despite higher and unpredictable latencies in long distance migrations.

Our clone process is implemented using a kernel thread that reads from the source disk and issues writes to the destination. This clone thread proceeds linearly across the disk, copying one disk region at a time. As usual, the remote write IOs are first enqueued to disk transmit buffer, then transferred to the destination host by the Streams framework, and then written to disk on the destination.

The clone process operates independently of the IO Mirror. This introduces a potential complication. A VM could issue a write while a read operation in the clone process is in progress, thus, the clone process could read an outdated version of the disk block. To avoid this, the IO Mirror module implements a synchronization mechanism to prevent clone process IOs and VM IOs from conflicting.

Synchronization works by classifying all VM writes into three types relative to our clone process: (1) writes to a region that has been copied (2) writes to a region being copied (3) writes to a region that has not yet been copied. No such synchronization is required for reads, which are always passed directly to the source disk.

For case (1), where we are writing to an already-copied region, any write must be mirrored to the destination to keep the disks in lock-step. Conversely, in case (3), where we are writing to a region that is still scheduled to be copied, the write does not need to be mirrored. The clone process will eventually copy the updated content when it reaches the write's target disk region.

More complex is case (2), where we receive a write to the region currently being copied by the clone process. In the case of local storage migration, where writes are synchronously mirrored, we defer the conflicting writes from the VM and place them into a queue. Once the clone process completes for that region, we issue the queued writes and unlock the region. Finally, we wait for in-flight IOs as we lock the next region, guaranteeing there are no active writes to our next locked region [21]. However, for long distance migration, network latency makes it prohibitively expensive to defer the conflicting VM IOs until the bulk copy IOs are complete. Instead, we introduce the concept of a *transmit snapshot*, a sequence of disk IOs captured with a sliding window in the transmit buffer.

As we receive IO, either clone or mirror IO, we fill the disk's active *transmit snapshot*. When the active *transmit snapshot* reaches capacity (1 MB by default), the snapshot is promoted to a *finalized snapshot*. This promotion process pushes the *transmit snapshot* window forward to cover the next region of the disk's transmit buffer, the new empty *transmit snapshot*. As soon as all of our clone process's IOs for a given region have been queued in a *transmit snapshot*, we unlock the copy region and move on to the next region.

The Streams framework searches for *finalized snapshots*, transmitting them in the order they are finalized. A write barrier is imposed between the transmission of each *finalized snapshot*, ensuring that snapshot content is not interleaved. Between these write barriers, source-side snapshot sector deduplication, and destination-side IO conflict chains, we ensure that all IOs are written to the destination in the correct order. See § 3.6 for more discussion of snapshot handling and IO conflict chains.

We perform source-side deduplication by coalescing repeated writes to the same disk block. This ensures correctness as blocks within a *transmit snapshot* may be reordered by Streams, and reduces bandwidth consumption. Upon receipt of new IO, we search our transmit snapshot for any IO to the same target disk sector. If there was a previous write, we coalesce the two into a single write of the most recent data. This is safe, as we know clone process IO and mirror IO will never conflict, as explained in our discussion of synchronization.

Scanning our transmit snapshot for duplicate disk IO can be expensive, as each disk's active snapshot could contain 1 MB worth of disk IO. We avoid such lookups with a bloom filter that tracks the disk offsets associated with each sector present in each disk's active snapshot. With 8 KB of memory dedicated to each disk's bloom filter, the disk sector as the key, and 8 hashes per key, we achieve a false positive rate of less than 6 per million blocks.

Our data showed that the use of the bloom filter and this optimization halved migration time and cut CPU usage by a factor of eight in some cases. This result is highly workload dependent, and the CPU utilization benefits of the bloom filter offer a large portion of the gains.

3.4 Memory and Disk Copy Coordination

There are several copying tasks for us to coordinate: copying our disk, IO mirroring, copying of the initial memory image, and iterative copying of memory data.

When we start and how we interleave these processes can have a significant impact. If we have insufficient bandwidth available to keep up with page dirtying, we may be forced to throttle the workload. If we do not efficiently use available bandwidth, we increase overall migration time. While low intensity workloads with relatively low dirty rates do not create significant contention for bandwidth, contention can become an issue with higher intensity workloads.

We begin our disk copy first as this is usually the bulk of our data, while enabling IO mirroring. For a while this "fills-the-pipe." After our clone process completes, we begin our initial memory copy, then begin our iterative memory copy process with IO Mirroring still enabled.

Our rationale for this is that storage mirroring generally requires lower bandwidth than memory copying, so overlapping these is less likely to lead to contention than if we try to perform our disk copy process while memory is being copied. Also, our disk copy generally takes longer to complete, thus, we prefer to start it first, to minimize the overall migration time.

Our intuition around memory tending to show an equal or higher dirtying rate, i.e., rate at which pages or disk blocks change, leading to a higher cost for our iterative copy than for IO mirroring, comes from our observation of databases workloads and other enterprise applications. Locks, metadata and other non-persistent structures appear to be a significant source of dirtying.

During the switchover time, where we switch from source to destination, we pause the VM for a short period to copy any remaining dirty memory pages and drain the storage buffers to disk. Draining time is a downside

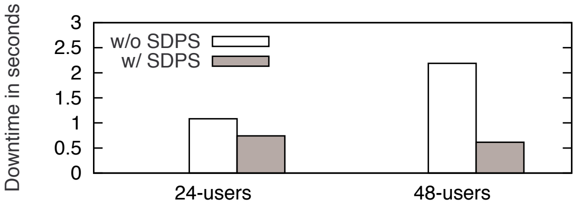


Figure 2: Shows downtime for two intensities of our DVD Store 2 workloads with and without SDPS. We show less downtime variation with SDPS than without.

of asynchronous mirroring as it impacts switchover time, which is not an issue for local synchronous mirroring.

While our heuristic approach seems to work well it may not be ideal. In an ideal implementation we would like to start the memory copy just before the disk copy has completed and only use unutilized bandwidth. Unfortunately, without predictable bandwidth it is hard to decide when we should begin the memory copy as bandwidth may suddenly become scarce, we could be forced to throttle our workload. We imagine using underlying network QoS mechanisms or better techniques for bandwidth estimation could be helpful.

3.5 Stun During Page Send (SDPS)

Transferring the VM's memory state is done using iterative pre-copy, as discussed in § 3.1. Iterative pre-copy converges when a VM's workload is changing page content (dirtying pages) more slowly than our transmission rate. Convergence means that the migration can trade-off longer migration time for ever smaller downtimes, and thus the migration can be completed without noticeable downtime. Unfortunately, this is not always the case, especially in long distance migrations.

Historically, live migration implementations have solved the problem of low bandwidth by stopping pre-copy, halting VM execution, and transmitting all remaining page content during VM downtime. This is not desirable, as the VM could remain halted for a long time if there are many dirty pages, or if the network is slow.

To solve this problem, we introduce Stun During Page Send (SDPS). SDPS monitors the VM's dirty rate, as well as the network transmit rate. If it detects that the VM's dirty rate has exceeded our network transmit rate, such that pre-copy will not converge, SDPS will inject microsecond delays into the execution of the VM's VCPUs. We impose delays in response to page writes, directly throttling the page dirty rate of any VCPU without disrupting the ability of the VCPU to service critical operations like interrupts. This allows XvMotion to throttle the page dirtying rate to a desired level, thus guaranteeing pre-copy convergence.

Figure 2 shows the downtime for our DVD Store 2 workload that is described in Section 5. We see less downtime variability when SDPS is turned on than without. This technique is critical to allowing live migration to support slower networks and handle intense workloads.

3.6 Disk Buffering Congestion Control

XvMotion supports migrating a VM with multiple virtual disks located on different source volumes to different destination volumes with potentially differing IO performance. For example, one virtual disk could be on a fast Fibre Channel volume destined for a slower NFS volume, and another on a slower local disk destined for a faster local SSD.

Disparities in performance introduce several challenges. For example, if we are moving a virtual disk from a faster source volume to a slower destination volume, and we have a single transmit queue shared with other virtual disks, the outgoing traffic for our slower destination can tie up our transmit queue, starving other destinations.

We address such cross-disk dependencies with queue fairness constraints. Each disk is allowed to queue a maximum of 16 MB worth of disk IO in the shared transmit queue. Attempts to queue additional IO are refused, and the IO is scheduled to be retried once the disk drops below the threshold. In effect, each virtual disk has its own virtual transmit queue.

Tracking per-disk queue length also allows us to solve the problem of the sender overrunning the receiver. To avoid sending too many concurrent IOs to the destination volume, we have a soft limit of 32 outstanding IOs (OIOs) per volume. If the number of OIOs on the destination exceeds this, the destination host can apply back pressure by requesting that the source host slow a given virtual disk's network transmit rate.

There is a final challenge the destination host must attend to. Since the source deduplicates the transmit snapshots by block number, we know that IOs within a given snapshot will never overlap, and can thus be issued in any order. However, there may be IO conflicts i.e., IOs to overlapping regions across snapshots. In such cases, it is important for correctness that IOs issue in the order that the snapshots are received. However, we don't want to limit performance by forcing all snapshots to be written in lockstep. Instead, we make the destination keep track of all in-flight IOs, at most 32 IOs for each disk. Any conflicting new IOs are queued until the in-flight IO is completed. This is implemented with a queue, called a conflict chain, that is associated with a given in-flight IO.

Fairness between clone and mirror IOs: We implement fairness between clone and mirror IOs in a similar way. Each disk's virtual queue is split into two, with one portion for the clone and the other for mirror IOs. These queues are drained in a weighted round-robin schedule. Without this, the guest workload and clone can severely impact one another when one is more intense than the other, possibly leading to migration failures.

4 Optimizations

4.1 Minimizing Switchover Time

The switchover time is the effective downtime experienced by the VM as we pause its execution context to move it between hosts. It is the phase we use to transfer the remaining dirty pages and the state of the guest's virtual devices (i.e., SVGA, keyboard, etc.). Today's applications can handle downtimes over 5 seconds, but an increasing number of high availability and media streaming applications can make sub-second downtimes noticeable. For these reasons, we implemented some optimizations to battle the following problems:

Large virtual device states: Some of the virtual devices can be very large, for example the SVGA (the video card buffer) can be hundreds of megabytes. The problem is that this state is not sent iteratively, so it has to be sent completely during downtime. Because of limited bandwidth at high latency, we implemented a solution where the state of the larger devices is stored as guest memory. Now, the state is sent iteratively, is subject to guest slowdown (i.e., SDPS), and, most importantly, is not sent in full during downtime.

TCP slow start: There are several TCP sockets used during downtime: some to transfer the state of the virtual devices, and some for the Streams transmission. The problem is that each time a TCP connection is used for the first time or after a delay, TCP is in slow start mode where it needs time to slowly open the windows before achieving full throughput (i.e., several hundreds of milliseconds). We implemented some TCP extensions where we set all the sockets windows to the last value seen before becoming idle. This change reduces many seconds of downtime at 100 ms of latency.

Synchronous RPCs: We found that our hot migration protocol had a total of 11 synchronous RPCs. At 100 ms round-trip time (RTT), this adds up to 1.1 seconds of downtime. After careful analysis, we concluded that most of these RPCs do not need to be synchronous. In fact, there are only 3 synchronization points needed during switchover: before and after sending the final set of dirty pages, and the final handshake that decides if the VM should run on the destination or the source (e.g., there was a failure).

We modified the hot migration protocol to only require these three round-trips. The idea was to make the RPCs asynchronous with the use of TCP's ordering and reliability guarantees. Specifically, we used the following 2 guarantees. (1) If the source sends a set of messages and the destination receives the last one, TCP ensures that it also received all the previous messages. (2) Moreover, if the source receives a reply to the last message, TCP ensures that the destination received all the previous messages. We added the three required synchronization points such that the source sends messages, but only waits for the reply of those three points.

4.2 Network Virtualization Challenges

To saturate a high latency TCP connection we must avoid inducing packet loss and reordering within the virtualized network stack. Due to network virtualization, a hypervisor's networking stack is complex, with multiple layers, memory heaps, and asynchronous contexts: all of them with their own queuing mechanisms and behavior when queues are full. When queue limits are exceeded, packet loss and reordering can occur, disrupting TCP streams.

We found that Virtual Network Switches on the source host were the main source of packet loss. Multiple XvMotion sockets with large buffers were constantly overflowing the virtual switches' queues, resulting in packet loss, which substantially impacted TCP performance. We tried adding Random Early Detection (RED [13]) to the Virtual Network Switch, which drops packets from the head of queues rather than the tail so the sender could detect packet loss sooner. However, we found that the resulting packet losses still significantly reduced throughput. The solution we settle upon was to allow virtual switches to generate back pressure when their queues were nearly full by sending an explicit notification to the sending socket(s). This allowed our sockets to check for possible overflows, stop growing the transmit windows, and ultimately avoid the drops.

A second problem is the reordering of packets. Packets are moved from a virtual switch queue into a NIC ring, and whenever the NIC is full, packets are re-queued into the virtual switch. We found this to be a common source of packet reordering. We noticed that a single packet reorder caused a performance hit from 113 Mbps to 76 Mbps for a 1 GB data transfer over a 100 ms RTT link. After fixing these problems, we can get full throughput at 1 Gbps and up to 250 ms of round-trip time. Tests also showed that we were able to achieve 45% of bandwidth utilization at a loss rate of 0.01%.

4.3 Tuning for Higher Latency

High latency networks require provisioning larger TCP send socket buffers. One restriction on our platform, however, is the strictness of resource management and the desire to minimize memory usage for any given VM migration. We satisfy both goals by detecting the round-trip-time between the source and destination host in the initial migration handshake, dynamically resizing our socket buffers to the bandwidth-delay-product.

In the course of experimenting with socket buffer resizing, we discovered some performance issues surrounding our congestion control algorithms. We switched to the CUBIC congestion control algorithm, which is designed for high bandwidth-delay product networks [15]. Experimentation showed that we should enable congestion control with Accurate Byte Counting [7]. This improved performance when leveraging various offload and packet coalescing algorithms. It helps grow the congestion window faster and handles delayed acknowledgments better.

4.4 Virtual Disk Copy Optimizations

XvMotion implements several storage optimizations to remain on par with live storage migration between locally accessible volumes. Local live storage migration makes heavy use of a kernel storage offload engine called the Data Mover (DM) [21].

The DM is tightly coupled with the virtual disk format, file system, and storage array drivers to perform optimizations on data movement operations. The DM's primary value is its ability to offload copy operations directly to the storage array. However, it also implements a number of file system specific optimizations that are important for storage migration. Because the DM only works with local storage we cannot use the DM, instead, we reimplemented these optimizations in XvMotion.

Metadata Transaction Batching: In VMFS, metadata operations are very expensive as they take both in-memory and on-disk locks. To avoid per-block metadata transactions, we query the source file system to discover the state of the next 64 FS blocks, then use that data to batch requests to the destination file system for all metadata operations.

Skipping Zero Block Reads: Each file in VMFS is composed of file system blocks, by default 1MB each. A file's metadata tracks the allocation and zero state of those blocks. Blocks may not be allocated, may be known to be zero-filled, or allocated but still to-be-zeroed (TBZ) blocks. By querying the FS metadata, we can skip blocks that are in one of these zero states.

Skip-Zero Writes: On the destination, if we know we will write over the entire contents of a file system block, we can have the file system skip performing its typical zeroing upon the first IO to any FS block. We bias writing to entire FS blocks when possible to leverage such skip-zero opportunities.

5 Evaluation

Downtime: All live migration technologies strive to achieve minimal downtime i.e., no perceptible service disruption. Our results show that XvMotion delivers consistent downtimes of ≈ 1 second—what the convergence logic targets—in the face of variable round trip latency (0-200 ms) and workload intensity (1-16 OIOs), as depicted in Figures 3 and 4.

Migration time: Our results show that our migration times are stable with respect to latency up to 200 ms—demonstrating that wide area migration need not be any more expensive than local area migration. XvMotion shows only a small increase in migration time beyond the sum of local memory and storage migration time.

Guest Penalty: Guest penalty as a percentage of the reduction of guest performance (IOPS or Operations/sec.) during migration should be minimized. Our results show that guest penalty is nearly constant with respect to latency, and only varies based on workload intensity.

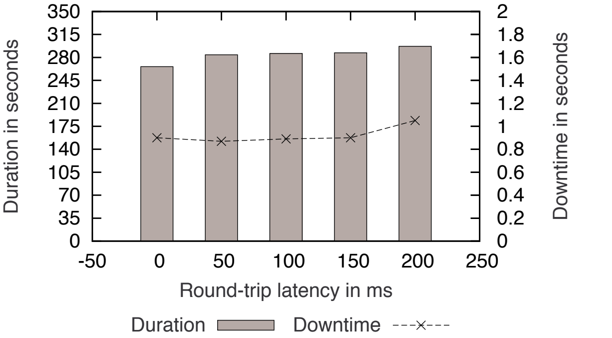


Figure 3: Shows the migration time and downtime for a VM running the OLTP workload at varying latency.

Our XvMotion tests were run on a pair of Dell R610 server running our modified version of ESX. Each had dual-socket six-core 2.67 GHz Intel Xeon X5650 processors, 48 GB of RAM, and 4 Broadcom 1 GbE network adapters. Both servers were connected to two EMC VNX5500 SAN arrays using an 8 Gb Fibre Channel (FC) switch. We created a 200 GB VMFS version 5 file system on a 15-disk RAID-5 volume on each array.

We used the Maxwell Pro network emulator to inject latency between the hosts. Our long distance XvMotions were performed on a dedicated link between Palo Alto, California and Banglore, India with a bottleneck bandwidth of 1 Gbps.

We used three workloads, an idle VM, OLTP Simulation using Iometer [5], and the DVD Store version 2 [3] benchmark. The Idle and Iometer VMs were configured with two vCPUs, 2 GB memory of RAM, and two virtual disks. The first disk was a 10 GB system disk running Linux SUSE 11 x64, the second was a 12 GB data disk.

During the migration both virtual disks were migrated. Our synthetic workload used Iometer to generate an IO pattern that simulates an OLTP workload with a 30% write, 70% read of 8 KB IO commands to the 12 GB data disk. In addition, we varied outstanding IOs (OIOs) to simulate workloads of differing intensities.

DVD Store Version 2.1 (DS2) is an open source online e-commerce test application, with a backend database component, and a web application layer. Our DVD Store VM running MS Windows Server 2012 (x64) was configured with 4 VCPUs, 8 GB memory, a 40 GB system/db disk, and a 5 GB log disk. Microsoft SQL Server 2012 was deployed in the VM with a database size of 5 GB and 10,000,000 simulated customers. Load was generated by three DS2 client threads with no think time.

5.1 Downtime (Switchover Time)

Figure 3 shows the downtimes for our OLTP workload with network round-trip latencies varying from 0-200 ms. All XvMotions show a bounded downtime of roughly one second.

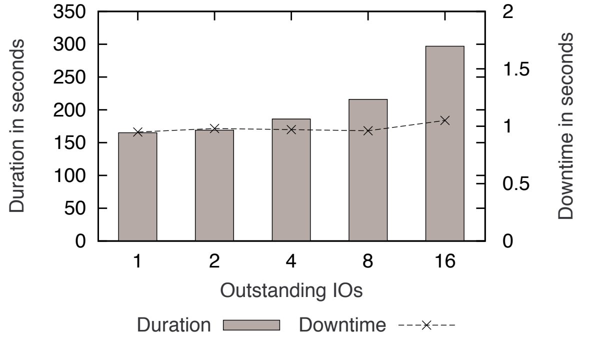


Figure 4: Migration time and downtime for varying OIO on the OLTP workload at 200 ms of round-trip latency.

The XvMotion protocol iteratively sends memory pages: at every iteration it sends the pages written by the guest in the previous iteration. It converges when the remaining data can be sent in half a second, which is the value used in these experiments. Thanks to our SDPS optimization, even if the network bandwidth is low and the guest is aggressively writing pages, the remaining data is bounded to be sent in half a second.

When we measured downtime versus storage workload intensity (varying OIO) we discovered that the switchover time remained nearly constant. Figure 4 shows an average downtime of 0.97 seconds, with a standard deviation of 0.03. This shows we have achieved downtime independent of storage workload.

5.2 Migration Time

We compared an XvMotion between two hosts over 10 Gbps Ethernet, to local live storage migration in Figure 5, to approximate a comparison between XvMotion and local live migration.

Local live storage migration copies a virtual disk between two storage devices on the same host. While this is a bit of an apples to oranges comparison, we choose it for a few reasons. First, storage migration overhead generally dominates memory migration overhead. Second, storage migration is quite heavily optimized, so it provides a good baseline. Third, initiating a simultaneous local memory and storage migration would still not provide an apples to apple comparison as there would be no contention between memory and storage migration on the network, and the copy operations are uncoordinated.

As expected, live storage migration is the fastest. The data shows about 20 to 23 seconds difference between local live storage migration and XvMotion. Part of that overhead is from memory and device state migration that required about 2 seconds to migrate memory state in the Idle and OLTP scenarios, and 8 seconds in DVD Store scenario. Additional reasons that these differences exist are contention between memory migration and IO mirroring traffic, and greater efficiencies in the data mover. However, the encouraging result here is that local migration between hosts is not appreciably more expensive than migrating storage from one volume to another on the same host.

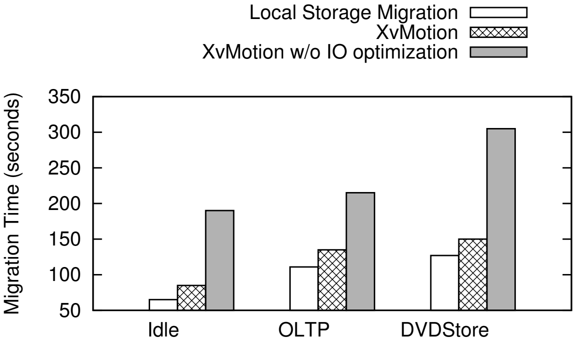


Figure 5: Shows the migration time for three workloads. The remote migration cases are approximately 10% slower for OLTP and DVDStore.

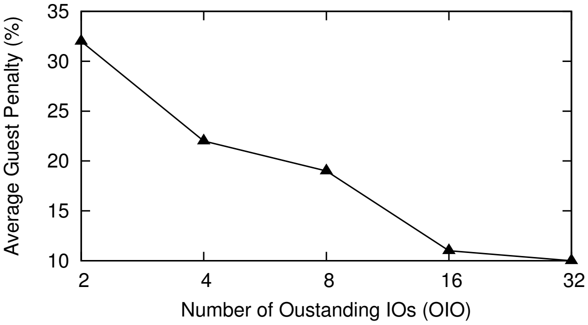


Figure 6: Guest workload penalty as workload intensity (OIO) increases. As workload intensity increases, the average guest penalty decreases, as our transmit queue scheduler gives increased priority to Guest IOs over clone process IOs. workload IOs over clone process IOs.

A last point illustrated by Figure 5 is the importance of our storage optimizations described in § 4.4. Migration time nearly doubles in all the scenarios without our disk copy optimizations.

In Figure 4 we measured migration time as we vary workload intensity. We see that migration time increases with guest OIO. This occurs because clone IO throughput decreases as guest IOs take up a larger fraction of the total throughput.

Figure 3 presents how the migration time of the OLTP workload changes as we vary round-trip time. We see that migration time increases by just 10% when increasing latency from 0 to 200 ms. We observed that most of the overhead came from the memory copy phase. In this case, TCP was not able to optimally share the 1 Gbps available bandwidth to both the memory and the disk mirror copy sockets. Additionally, TCP slow start at the beginning of every phase of the migration also add several seconds to the migration time.

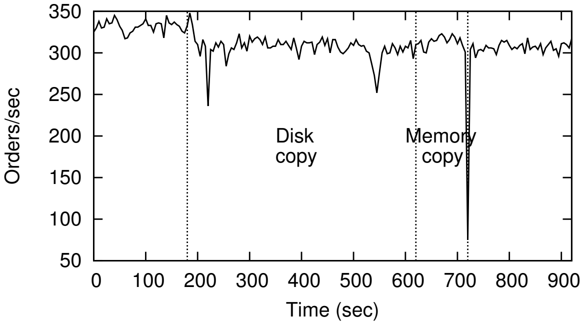


Figure 7: Illustrates the various phases of the migration against a plot of DVD Store Orders/sec for XvMotion.

5.3 Guest Penalty

Figure 6 shows average guest penalty decreases as workload intensity increases. This ensures that the impact of migration on guest performance is minimized.

The average guest penalty drops from 32% for 2 OIOs to 10% for 32 OIOs. This is because both clone process and mirror IOs compete for transmit queue space on the source as discussed in § 3.6. The clone process has at most 16 OIOs on the source to read disk blocks for the bulk copy. As the guest workload has more OIOs it will take a larger share of transmit queue.

Our transmit queue schedules bandwidth between clone process and mirror IOs. Without such a scheduler, we measured a 90% average guest penalty for the 8 OIO scenario. To further investigate, we measured the effect of varying latency for a fixed intensity workload. We observed the penalty for the OLTP workload with 16 OIO is fixed at approximately 10% for RTT latencies of 0 ms, 100 ms and 200 ms.

Figure 7 shows a dissection of DVD Store's reported workload throughput during an XvMotion on a 1 GbE network link with 200 ms round-trip latency. The graph shows the impact on the SQL server throughput during disk and memory copy, and shows a quick dip for one data point as the VM is suspended for switchover. This is an example of a successful migration onto a slower destination volume, where the workload is gradually throttled to the speed of the destination.

5.4 XvMotion on the Wide Area

We explored XvMotion's behavior on the wide area by migrating a VM from Palo Alto, CA to Bangalore, India. These two sites, about halfway across the globe, are separated by a distance of roughly 8200 miles. The WAN infrastructure between these two sites was supported by a Silverpeak WAN accelerator NX 8700 appliance. The measured ping latency between the sites was 214 ms over a dedicated 1 Gbps link.

Our source host in Palo Alto was a Dell PowerEdge R610 with a high performance SAN storage array. Our destination in Bangalore was a Dell PowerEdge R710 with local storage. To virtualize the network, we stretched the layer-2 LAN across these two sites using an OpenVPN bridge. This enabled the VM migrate with no disruption in network connectivity.

We successfully ran two workloads, Iometer and DS2, with minimal service disruption.

In the Iometer test, before the migration, the ping latency between the client and the OLTP VM was less than 0.5 ms since both were running on the same local site. After the migration the ping latency between the client and the OLTP VM jumped to 214 ms. The ping client observed just a single ping packet loss during the entire migration.

The OLTP workload continued to run without any disruption, although the guest IOPS dropped from about 650 IOPS before migration to about 470 IOPS after migration as the destination host was using low performance local storage.

The total duration of the migration was 374.16 seconds with a downtime of 1.395 seconds. We observed 68.224 MB/s network bandwidth usage during disk copy, and about 89.081 MB/s during memory pre-copy.

In the DS2 test, the DS2 clients ran on two client machines, one client machine located in Palo Alto and the other in Bangalore. The DS2 clients generated a substantial CPU and memory load on the VM. The migration was initiated during the steady-state period of the benchmark, when the CPU utilization of the virtual machine was little over 50%.

The DS2 workload continued to run without any disruption after the migration. However, the throughput dropped from about 240 ops before migration to about 135 ops after migration as the destination host was using low performance local storage.

The total duration of the migration was 1009.10 seconds with a downtime of 1.393 seconds. We observed 49.019 MB/s network bandwidth usage during disk copy, and about 73.578 MB/s during memory pre-copy. Some variation in network bandwidth was expected between the tests as the WAN link between sites is a shared link.

5.5 Summary

Our results show that XvMotion exhibits consistent behavior i.e., migration time, downtime, and guest penalty, with varying workload intensity and latency. We saw for all experiments, maximum downtime is around one second, even for latencies as high as 200 ms and data loss up to 0.5%. One of the nice features of our system is that for long distance migrations the VM migration was bottlenecked by the network bandwidth, and thus our storage throttling mechanism slowed the VM down gradually.

6 Discussion

We discuss potential split-brain issues when handing off execution from the source VM to the destination VM, then briefly survey networking and security considerations for wide area migration.

6.1 Split-Brain Issues

A live migration ends with the exchange of several messages between the source and destination hosts called the resume handshake. In the normal case, if the source receives the resume handshake, it can mark the migration successful, and power off. Upon sending the resume handshake, the destination can resume and mark the migration complete. With migration using shared storage, file system locking ensures only one VM can start during a network partition. Each host will race to grab locks on the virtual disks and metadata file, with only one winner.

With XvMotion there is no good solution and yet we must prevent both hosts from resuming concurrently. Both the source and destination wait for the completion of the resume handshake. The source will power on if it never sees a resume request. A network partition after the resume request message causes both hosts to power-off. If the destination receives approval to resume, it will from that point on. Finally, when the source receives the resume completed message it will power-off and cleanup. Anytime the hosts cannot resolve the issue, a human or the management software must power on the correct VM and delete the other VM.

6.2 VM Networking

Long distance migration presents a challenge for the VM's network connectivity. A VM may be migrated away from its home subnet and require a form of network tunneling or routing. Several classes of solutions are available to enable long distance network migrations. First, there are layer two networking solutions that tunnel traffic using hardware or software solutions such as Cisco Overlay Transport Virtualization (OTV) [14] and VXLAN [20]. For modern applications that do not depend on layer two connectivity there are several layer 3 solutions such as Locator/ID Separation Protocol (LISP) [12] that enable IP mobility across the Internet. Another emerging standard being deployed at data centers is OpenFlow [22], which enables generic programming and configuration of switching hardware. Using OpenFlow, several prototypes have been constructed that enable long distance VM migration.

6.3 Security Considerations

Any live migration technology introduces a security risk as a VM's memory and disk are transmitted over the network. Typically, customers use physically or logically isolated networks for live memory migration, but this is not sufficient for migrations that may be over the WAN. Today customers address this through the use of hardware VPN solutions or IPSec. While some customers may desire other forms of over the wire encryption support, we regarded this as outside the scope of our current work.

7 Related Work

Live Migration: Live VM memory migration has been implemented in all major hypervisors including Xen [10], Microsoft Hyper-V [1], and VMware ESX [24]. All three systems use a pre-copy approach to iteratively copy memory pages from source host to destination host.

Hines *et al.* [18] proposed a post-copy approach for live VM memory migration. Their approach essentially flips the steps of an iterative copy approach, instead of sending the working set at the end of the migration, it is sent at the beginning. This allows execution to immediate resume on the destination, while memory pages are still being pushed, and missing pages are demand paged in over the network. Memory ballooning is used to reduce the size of the working set prior to migration. Post-copying offers lower migration times and downtimes, but often induces a higher guest penalty and gives up atomic switch over. Luo *er al.* [19] used a combination of post-copy and pre-copy approaches to lower downtime downtime and guest penalty, but also gives up atomicity. Both of these approaches are unacceptable for wide area migration because of increased risk to losing the VM. Our SDPS technique offers a safer approach, reducing downtime without the loss of atomicity.

Storage Migration: ESX live storage migration has evolved through three different architectures, i.e., snapshotting, dirty-block tracking and IO mirroring [21]. Live storage migration of VMs using IO mirroring is explored in Meyer et al. [23]. The latest storage migration implementation in Microsoft Hyper-V [4], VMware ESX, and Xen with DRBD [6], are all based on IO mirroring.

WAN Migration: Bradford *et al.* extends the live migration in Xen to support the migration of a VM with memory and storage across the WAN [9]. When a VM is being migrated, its local disks are transferred to destination volume using a disk block level iterative pre-copy. The write IO workload from the guest OS is also throttled to reduce the dirty block rate. Further optimizations for pre-copy based storage migration over WAN are explored by Zheng *et al.* [28].

While the iterative pre-copy approach is well suited for memory migration, it suffers from several performance and reliability limitations for storage migration as shown in our prior work [21]. In contrast, we propose to seamlessly integrate memory pre-copy with storage IO mirroring for long distance live VM migration.

CloudNet addresses many of the shortcomings of Bradford's work by using DRBD and Xen to implement wide area migration along with a series of optimizations [27]. The system used synchronous disk replication rather than asynchronous replication used by XvMotion. Their *Smart Stop and Copy* algorithm tuned the number of iterations for memory copy, thus trading off downtime versus migration time. ESX used a similar algorithm internally, but downtimes were still sufficiently high even with this measure that we introduced SDPS. SDPS and asynchronous disk buffering allows XvMotion to target a specific downtime at the cost of increased guest penalty.

SecondSite is the first solution to use software fault tolerance to implement seamless failover of a group of VMs over a WAN [26]. This solution is built on Remus [11] a fault tolerance solution built on Xen's live migration infrastructure. SecondSite and Remus provide the destination periodically with consistent images of the VMs memory and disk. This is done while the VM is running on the source and the migration only completes when the source host dies.

8 Conclusion

We have presented XvMotion, a system for memory and storage migration over local and wide area networks. By integrating memory and storage movement we were able to achieve an atomic switchover with low downtime. Our use of asynchronous storage replication provides good storage performance in the presence of high latency links. We also introduced mechanisms to increase memory migration tolerance to high latency links, and make storage migration robust to diverse storage speeds.

Our OLTP tests show that an XvMotion between two separate hosts over 10 Gbps Ethernet, performed only 10% slower than a storage migration on a single host between two locally attached disks, demonstrating live migration on a shared-nothing architecture is comparable to live migration with shared storage. We also showed that while increasing the latency of the network to 200 ms we saw downtimes lower than one second, which are unnoticeable to most applications, demonstrating the live migration is viable over the wide area. We also showed that our system is well behaved under heavy load, as increases in guest workload do not effect downtime.

Higher bandwidth networks, network virtualization, large scale virtualization deployments, geographically separated data centers and diverse storage architectures are all increasingly important parts of data centers. Given these trends, we believe the ability to simply, reliably, and efficiently move a VM between two hosts afforded by XvMotion will enable new use cases, and help simplify existing situations where VM mobility is demanded.

9 Acknowledgements

We would like to thank all the people who gave us feedback and support: Michael Banack, Dilpreet Bindra, Bruce Herndon, Stephen Johnson, Haripriya Rajagopal, Nithin Raju, Mark Sheldon, Akshay Sreeramoju, and Santhosh Sundararaman. This work was partially funded by DARPA CRASH grants N66001-10-2-4088 and N66001-10-2-4089.

References

[1] Windows Server 2008 R2 Hyper-V Live Migration, June 2009. `http://www.microsoft.com/en-us/download/details.aspx?id=12601`.

[2] Distributed Replicated Block Device (DRBD), May 2012. `http://www.drbd.org/`.

[3] DVD Store version 2 (DS2), May 2012. `http://en.community.dell.com/techcenter/extras/w/wiki/dvd-store.aspx`.

[4] How does Storage Migration actually work?, Mar. 2012. `http://blogs.msdn.com/b/virtual_pc_guy/archive/2012/03/14/how-does-storage-migration-actually-work.aspx`.

[5] Iometer, May 2012. `http://www.iometer.org/`.

[6] Chapter 13: Using Xen with DRBD, May 2014. `http://www.drbd.org/users-guide/ch-xen.html`.

[7] ALLMAN, M. TCP Congestion Control with Appropriate Byte Counting (ABC), 2003.

[8] ASPESI, J., AND SHOREY, O. EMC VPLEX Metro Witness Technology and High Availability. Tech. Rep. h7113-vplex-architecture-deployment.pdf, EMC, Mar. 2012.

[9] BRADFORD, R., KOTSOVINOS, E., FELDMANN, A., AND SCHIÖBERG, H. Live Wide-Area Migration of Virtual Machines Including Local Persistent State. VEE '07.

[10] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. NSDI'05.

[11] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 161–174.

[12] FARINACCI, D., FULLER, V., MEYER, D., AND LEWIS, D. Locator/ID Separation Protocol (LISP). Tech. rep., IETF, Aug. 2012. Work in progress.

[13] FLOYD, S., AND JACOBSON, V. Random Early Detection Gateways for Congestion Avoidance. *Networking, IEEE/ACM Transactions on 1*, 4 (1993), 397–413.

[14] GROVER, H., RAO, D., FARINACCI, D., AND MORENO, V. Overlay Transport Virtualization. Internet-Draft draft-hasmit-otv-03, Internet Engineering Task Force, July 2011. Work in progress.

[15] HA, S., RHEE, I., AND XU, L. CUBIC: A New TCP-friendly High-speed TCP Variant. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 64–74.

[16] HACKER, T. J., ATHEY, B. D., AND NOBLE, B. The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM* (2002), IEEE, pp. 434–443.

[17] HACKER, T. J., NOBLE, B. D., AND ATHEY, B. D. Improving throughput and maintaining fairness using parallel tcp. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies* (2004), vol. 4, IEEE, pp. 2480–2489.

[18] HINES, M. R., DESHPANDE, U., AND GOPALAN, K. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev. 43*, 3 (2009), 14–26.

[19] LUO, Y. A three-phase algorithm for whole-system live migration of virtual machines. CHINA HPC '07.

[20] MAHALINGAM, M., DUTT, D., AND ETC. VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. Tech. rep., IETF, Feb. 2012. Work in progress.

[21] MASHTIZADEH, A., CELEBI, E., GARFINKEL, T., AND CAI, M. The Design and Evolution of Live Storage Migration in VMware ESX. USENIX ATC'11.

[22] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev. 38*, 2 (Mar. 2008), 69–74.

[23] MEYER, D. T., CULLY, B., WIRES, J., HUTCHINSON, N. C., AND WARFIELD, A. Block Mason. In *Proceedings of the First Conference on I/O Virtualization* (Berkeley, CA, USA, 2008), WIOV'08, USENIX Association, pp. 4–4.

[24] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast Transparent Migration for Virtual Machines. USENIX ATC'05.

[25] PEDDEMORS, A., SPOOR, R., DEKKERS, P., AND BESTEN, C. Using drbd over wide area networks. Tech. Rep. drbd-vmigrate-v1.1.pdf, SurfNet, Mar. 2011.

[26] RAJAGOPALAN, S., CULLY, B., O'CONNOR, R., AND WARFIELD, A. SecondSite: Disaster Tolerance As a Service. *SIGPLAN Not. 47*, 7 (Mar. 2012), 97–108.

[27] WOOD, T., RAMAKRISHNAN, K., VAN DER MERWE, J., AND SHENOY, P. CloudNet: A Platform for Optimized WAN Migration of Virtual Machines. *University of Massachusetts Technical Report TR-2010-002* (January 2010).

[28] ZHENG, J., NG, T. S. E., AND SRIPANIDKULCHAI, K. Workload-aware live storage migration for clouds. VEE '11.

GPUvm: Why Not Virtualizing GPUs at the Hypervisor?

Yusuke Suzuki
Keio University

Shinpei Kato
Nagoya University

Hiroshi Yamada
Tokyo University of Agriculture and Technology

Kenji Kono
Keio University

Abstract

Graphics processing units (GPUs) provide orders-of-magnitude speedup for compute-intensive data-parallel applications. However, enterprise and cloud computing domains, where resource isolation of multiple clients is required, have poor access to GPU technology. This is due to lack of operating system (OS) support for virtualizing GPUs *in a reliable manner*. To make GPUs more mature system citizens, we present an open architecture of GPU virtualization with a particular emphasis on the Xen hypervisor. We provide design and implementation of full- and para-virtualization, including optimization techniques to reduce overhead of GPU virtualization. Our detailed experiments using a relevant commodity GPU show that the optimized performance of GPU para-virtualization is yet two or three times slower than that of pass-through and native approaches, whereas full-virtualization exhibits a different scale of overhead due to increased memory-mapped I/O operations. We also demonstrate that coarse-grained fairness on GPU resources among multiple virtual machines can be achieved by GPU scheduling; finer-grained fairness needs further architectural support by the nature of non-preemptive GPU workload.

1 Introduction

Graphics processing units (GPUs) integrate thousands of compute cores on a chip. Following a significant leap in hardware performance, recent advances in programming languages and compilers have allowed compute applications, as well as graphics, to use GPUs. This paradigm shift is often referred to as general-purpose computing on GPUs, *a.k.a.*, GPGPU. Examples of GPGPU applications include scientific simulations [26,38], network systems [13,17], file systems [39,40], database management systems [14,18,22,36], complex control systems [19,33], and autonomous vehicles [15,27].

While significant performance benefits of GPUs have attracted a wide range of applications, main governors of practically deployed GPU-accelerated systems, however, are limited to non-commercial supercomputers. Most GPGPU applications are still being developed in the research phase. This is largely due to the fact that GPUs and their relevant system software are not tailored to support virtualization, preventing enterprise servers and cloud computing services from isolating resources on multiple clients. For example, Amazon Elastic Compute Cloud (EC2) [1] employs GPUs as computing resources, but each client is assigned with an individual physical instance of GPUs.

Current approaches to GPU virtualization are classified into I/O pass-through [1], API remoting [8, 9, 11, 12, 24, 37, 41], or hybrid [7]. These approaches are also referred to as *back-end*, *front-end*, and *para* virtualization, respectively [7]. I/O pass-through, which exposes GPU hardware to guest device drivers, can minimize overhead of virtualization, but the owner of GPU hardware is limited to a specific VM by hardware design.

API remoting is more oriented to multi-tasking and is relatively easy to implement, since it needs only to export API calls to outside of guest VMs. Albeit simple, this approach lacks flexibility in the choice of languages and libraries. The entire software stack must be rewritten to incorporate an API remoting mechanism. Implementing API remoting could also result in enlarging the trusted computing base (TCB) due to accomodation of additional libraries and drivers in the host.

The para-virtualization allows multiple VMs to access the GPU by providing an ideal device model through the hypervisor, but guest device drivers must be modified to support the device model. According to these three classes of approaches, it is difficult, if not impossible, to use vanilla device drivers for guest VMs while providing resource isolation on multiple VMs. This lack of reliable virtualization support prevents GPU technology from the enterprise market. In this work, we explore GPU virtualization that allows multiple VMs to share underlying GPUs without modification of existing device drivers.

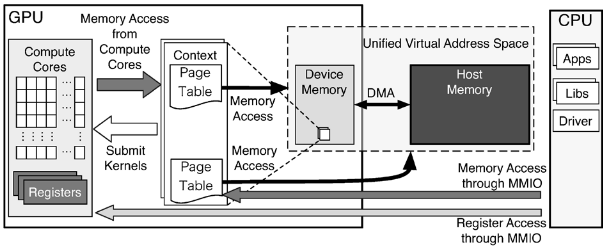


Figure 1: The GPU resource management model.

Contribution: This paper presents GPUvm, which is an open architecture of GPU virtualization. We provide the design and implementation of GPUvm based on the Xen hypervisor [3], introducing virtual memory-mapped I/O (MMIO), GPU shadow channels, GPU shadow page tables, and virtual GPU schedulers. These pieces of resource management are provided with both full- and para-virtualization approaches by exposing a native GPU device model to guest device drivers. We also develop several optimization techniques to reduce overhead of GPU virtualization. To the best of our knowledge, this is the first piece of work that addresses fundamental problems of GPU virtualization. GPUvm is provided as complete open-source software [1].

Organization: The rest of this paper is organized as follows. Section 2 describes a system model behind this paper. Section 3 provides a design concept of GPUvm, and Section 4 presents its prototype implementation. Section 5 shows experimental results. Section 6 discusses related work. This paper concludes in Section 7.

2 Model

The system is composed of a multi-core CPU and an onboard GPU connected on the bus. A compute-intensive function offloaded from the CPU to the GPU is called a *GPU kernel*, which could produce a large number of compute threads running on a massive set of compute cores integrated in the GPU. The given workload may also launch multiple kernels within a single process.

Product lines of GPU vendors are closely tied with programming languages and architectures. For example, NVIDIA invented the Compute Unified Device Architecture (CUDA) as a GPU programming framework. CUDA was first introduced in the Tesla architecture [30], followed by the Fermi and the Kepler architectures [30,31]. The prototype system of GPUvm presented in this paper assumes these NVIDIA technologies, yet the design concept of GPUvm is applicable for other architectures and programming languages.

Figure 1 illustrates the GPU resource management model, which is well aligned with, but is not limited to, the NVIDIA architectures. The detailed hardware mechanism is not identical among different vendors, though recent GPUs adopt the same high-level design presented in Figure 1.

MMIO: The current form of GPU is an independent compute device. Therefore the CPU communicates with the GPU via MMIO. MMIO is the only interface that the CPU can directly access the GPU, while hardware engines for direct memory access (DMA) are supported to transfer a large size of data.

GPU Context: Just like the CPU, we must create a context to run on the GPU. The context represents the state of GPU computing, a part of which is managed by the device driver, and owns a virtual address space in GPU.

GPU Channel: Any operation on the GPU is driven by commands issued from the CPU. This command stream is submitted to a hardware unit called a GPU *channel*, and isolated from the other streams. A GPU channel is associated with exactly one GPU context, while each GPU context can have one or more GPU channels. Each GPU context has GPU channel descriptors for the associated hardware channels, each of which is created as a memory object in the GPU memory. Each GPU channel descriptor stores the settings of the corresponding channel, which includes a *page table*. The commands submitted to a GPU channel is executed in the associated GPU context. For each GPU channel, a dedicated command buffer is allocated in the GPU memory visible to the CPU through MMIO.

GPU Page Table: Paging is supported by the GPU. The GPU context is assigned with the GPU page table, which isolates the virtual address space from the others. A GPU page table is set to a GPU channel descriptor. All the commands and programs submitted through the channel are executed in the corresponding GPU virtual address space.

GPU page tables can translate a GPU virtual address to not only a GPU device physical address but also a host physical address. This means that the GPU virtual address space is unified over the GPU memory and the host main memory. Leveraging GPU page tables, the commands executed in the GPU context can access to the host physical memory with the GPU virtual address.

PCIe BAR: The following information includes some details about the real system. The host computer is based on the x86 chipset and is connected to the GPU upon the PCI Express (PCIe). The base address registers (BARs) of PCIe, which work as windows of MMIO, are configured at boot time of the GPU. GPU control registers and GPU memory apertures are mapped on the BARs, allowing the device driver to configure the GPU and access the GPU memory.

To operate DMA onto the associated host memory,

[1] `https://github.com/CS005/gxen`

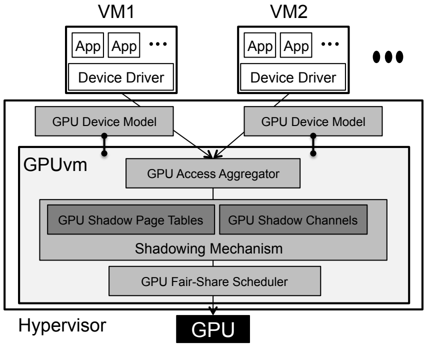


Figure 2: The design of GPUvm and system stack.

GPU has a mechanism similar to IOMMU, such as the graphics address remapping table (GART). However, since DMA issued from the GPU context goes through the GPU page table, the safety of DMA can be guaranteed without IOMMU.

Documentation: Currently GPU vendors hide the details of GPU architectures due to a marketing reason. Implementations of device drivers and runtime libraries are also protected by binary proprietary software, whereas the compiler source code has been recently open-released from NVIDIA to a limited extent. Some work uncovers the black-boxed interaction between the GPU and the driver [28]. Recently the Linux kernel community has developed Nouveau [25], which is an open-source device driver for NVIDIA GPUs. Throughout their development, the details of NVIDIA architectures are well documented in the Envytools project [23]. Interested readers are encouraged to visit their website.

Scope and Limitation: GPUvm is an open architecture of GPU virtualization with a solid design and implementation using Xen. Its implementation focuses on NVIDIA Fermi- and Kepler-based GPUs with CUDA. We also restrict our attention to Nouveau as a guest device driver. NVIDIA binary drivers should be available with GPUvm, but they cannot be successfully loaded with current versions of Xen, even in the pass-through mode, which has also happened in the prior work [11, 12].

3 Design

The challenge of GPUvm is to show that GPU can be virtualized at the hypervisor level. GPU is a unique and complicated device and its resources (such as memory, channels and GPU time) must be multiplexed like the host computing system. Although the architectural details of GPU are not known well, GPUvm virtualizes GPUs by combining the well-established techniques of CPU, memory, and I/O virtualization of the traditional hypervisors.

Figure 2 shows the high-level design of GPUvm and relevant system stack. GPUvm exposes a native GPU device model to VMs where guest device drivers are loaded. VM operations upon this device model are redirected to the hypervisor so that VMs can never access the GPU directly. The GPU Access Aggregator arbitrates all accesses to the GPU to partition GPU resources across VMs. GPUvm adopts a client-server model for communications between the device model and the aggregator. While arbitrating accesses to the GPU, the aggregator modifies them to manage the status of the GPU. This mechanism allows multiple VMs to access a single GPU in the isolated way.

3.1 Approaches

GPUvm enables full- and para-virtualization of GPUs at the hypervisor. To isolate multiple VMs on the GPU hardware resources, memory areas, PCIe BARs, and GPU channels must be multiplexed among those VMs. In addition to this spacial multiplexing, the GPU also needs to be scheduled in a fair-share manner. The main components of GPUvm to address this problem include GPU shadow page tables, GPU shadow channels, and GPU fair-share schedulers.

To aggregate accesses to a GPU device model from a guest device driver, GPUvm intercepts MMIO by setting these ranges as inaccessible.

In order to ensure that one VM can never access the memory area of another VM, GPUvm creates a GPU *shadow* page table for every GPU channel descriptor, which is protected from guest OSes. All GPU memory accesses are handled by GPU shadow page tables; a virtual address for GPU memory is translated by the shadow page table not by the one set by the guest device driver. Since GPUvm validates the contents of shadow page tables, GPU memory can be safely shared by multiple VMs. And by making use of GPU shadow page tables, GPUvm guarantees that DMA initiated by GPU never accesses memory areas outside of those allocated to the VM.

To create a GPU context, the device driver must establish the corresponding GPU channel. However, the number of GPU channels is limited in hardware. To multiplex VMs on GPU channels, GPUvm creates *shadow* channels. GPUvm configures shadow channels, assigns virtual channels to each VM and maintains the mapping between a virtual channel and a shadow channel. When guest device drivers access a virtual channel assigned by GPUvm, GPUvm intercepts and redirects the operations to a corresponding shadow channel.

3.2 Resource Partitioning

GPUvm partitions physical memory space and MMIO space over PCIe BARs into multiple sections of continuous address space, each of which is assigned to an individual VM. Guest device drivers consider that physical memory space origins at 0, but actual memory access is shifted by the corresponding size through shadow page tables created by GPUvm. Similarly, PCIe BARs and GPU channels are partitioned by multiple sections of the same size for individual VMs.

The static partitioning is not a critical limitation of GPUvm. Dynamic allocation is possible. When a shadow page table refers to a new page, GPUvm allocates a page, assigns it to a VM and maintains the mappings between guest physical GPU pages and host physical GPU pages. For ease of implementation, the current GPUvm employs static partitioning. We plan to implement the dynamic allocation in the future.

3.3 GPU Shadow Page Table

GPUvm creates GPU shadow page tables in the reserved area of GPU memory, which translates guest GPU virtual addresses to GPU device physical or host physical addresses. By design, the device driver needs to flush TLB caches every time a page table entry is updated. GPUvm can intercept TLB flush requests because those requests are issued from the host CPU through MMIO. After the interception, GPUvm updates the corresponding GPU shadow page table entry.

GPU shadow page tables play an important role in protecting GPUvm itself, shadow page tables, and GPU contexts from buggy or malicious VMs. GPUvm excludes the memory mappings to those sensitive memory pages from the shadow page tables. Since all the memory accesses by GPU go through the shadow page tables, any VMs cannot access those sensitive memory areas.

There is a subtle problem regarding a pointer to a shadow page table. In case of GPU, a pointer to a shadow page table is stored in GPU memory as part of the GPU channel descriptor. In the traditional shadow page tables, a pointer to a shadow page table is stored in a privileged register (e.g. CR3 in Intel x86); VMM intercepts the access to the privileged register to protect the shadow page tables. To protect GPU channel descriptors, including a pointer to a shadow page table, from being accessed by GPU, GPUvm excludes the memory areas for GPU channel descriptors from the shadow page tables. The accesses to the GPU channel descriptors from host CPUs are all through MMIO and thus, can be easily detected by GPUvm. As a result, GPUvm protect GPU channel descriptors, including pointers to shadow page tables, from buggy VMs.

GPUvm guarantees the safety of DMA. If a buggy driver sets an erroneous physical address when initiating DMA, the memory regions assigned to other VMs or the hypervisor can be destroyed. To avoid this situation, GPUvm makes use of shadow page tables and the unified memory model of GPU. As explained in Section 2, GPU page tables can map GPU virtual addresses to physical addresses in GPU memory and host memory. Unlike conventional devices, GPU uses GPU virtual addresses to initiate DMA. If the mapped memory happens to be in the host memory, DMA is initiated. Since shadow page tables are controlled by GPUvm, the memory access by DMA is confined in the memory region of the corresponding VM.

The current design of GPU poses an implementation problem of shadow page tables. In the traditional shadow page tables, page faults are extensively utilized to reduce the cost of constructing shadow page tables. However, GPUvm cannot handle page faults caused by GPU [10]. This makes it impossible to update the shadow page table upon page fault handling, and it is also impossible to trace changes to the page table entry. Therefore, GPUvm scans the entire page tables upon TLB flush.

3.4 GPU Shadow Channel

The number of GPU channels is limited in hardware and they are numerically indexed. The device driver assumes that these indexes start from zero. Since the same index cannot be assigned to multiple channels, channel isolation must be supported to multiplex VMs.

GPUvm provides GPU shadow channels to isolate GPU accesses from VMs. Physical indexes of GPU channels are hidden from VMs but virtual indexes are assigned to their virtual channels. Mapping between physical and virtual indexes is managed by GPUvm.

When a GPU channel is used, it must be activated. GPUvm manages currently activated channels by VMs. These activation requests are submitted through MMIO and they can be intercepted by GPUvm. When GPUvm receives the requests, GPUvm activates the corresponding channels by using physical indexes.

Each GPU channel has channel registers, through which the host CPU submits commands to GPU. Channel registers are placed in GPU virtual address space which is mapped to a memory aperture. GPUvm manages all physical channel registers and maintains the mapping between physical and virtual GPU channel registers. Since the virtual channel registers are mapped to the memory aperture, GPUvm can intercept the access to them and redirect it to the physical channel registers. Since the guest GPU driver can dynamically change the location of the channel registers, GPUvm monitors it and changes the mapping if necessary.

3.5 GPU Fair-Share Scheduler

So far we have argued virtualization of memory resources and GPU channels for multiple VMs. We herein provide virtualization of GPU time. Indeed this is a scheduling problem. The GPU scheduler of GPUvm is based on the bandwidth-aware non-preemptive device

(BAND) scheduling algorithm [21], which was developed for virtual GPU scheduling. The BAND scheduling algorithm is an extension of the CREDIT scheduling algorithm [3] in that (i) the prioritization policy uses reserved bandwidth and (ii) the scheduler intentionally inserts a certain amount of waiting time after completion of GPU kernels, which leads to fairer utilization of GPU time among VMs. Since the current GPUs are not preemptive, GPUvm waits for GPU kernel completion and assigns credits based on the GPU usage. More details can be found in [21].

The BAND scheduling algorithm assumes that the total utilization of virtual GPUs could reach 100%. This is a flaw because there must be some interval that the CPU executes the GPU scheduler during which the GPU remains idle, causing utilization of the GPU to be less than 100%. This means that even though the total bandwidth is set to 100%, VMs' credit would be left unused, if the GPU scheduler consumes some time in the corresponding period. The problem is that the amount of credit to be replenished and the period of replenishment are fixed. If the fixed amount of credit is always replenished, after a while all VMs could have a lot of credit left unused. As a result, the credit may not influence the decision of scheduling at all. To overcome this problem, GPUvm accounts for CPU time consumed by the GPU scheduler, and considers it as GPU time.

Note that there is a critical problem in guaranteeing the fairness of GPU time. If a malicious or buggy VM starts infinite computation on GPU, it can monopolize GPU time. One possible solution to this problem is to abort the GPU computation if the GPU time exceeds the pre-defined limit of the computation time. Another approach is to cut longer requests into smaller pieces, as shown in [4]. For the future directions, we are planning to incorporate the disengaged scheduling [29] at the VMM level. The disengaged scheduling provides fair, safe and efficient OS-level management of GPU resources. We believe that GPUvm can incorporate the disengaged scheduling without any technical issues except for engineering efforts.

3.6 Optimization Techniques

Lazy Shadowing: In principle, GPUvm has to reflect the contents of guest page tables to shadow page tables every time it detects TLB flushes. As explained in Section 3.3, GPUvm has to scan the entire page table to find the modified entries in the guest page table because GPUvm cannot use page faults to detect the modifications on the guest page tables. Since TLB flushes can happen frequently, the cost of scanning page tables introduces significant overhead. To reduce this overhead, GPUvm delays the reflection to the shadow page tables until an attempt is made to reference them. The shadow page tables are used when memory apertures are accessed or after the GPU kernels starts. Note that GPUvm can intercept memory apertures access and command submission. So, GPUvm scans the guest page table at this point of time and reflects it to the shadow page table. By delaying the reflection, GPUvm can reduce the number of the page table scans.

BAR Remap: GPUvm intercepts data accesses through BARs to virtualize GPU channel descriptors. By intercepting all data accesses, it keeps the consistency between shadow GPU channel descriptors and guest GPU channel descriptors. However, this design incurs non-trivial overheads because the hypervisor is invoked every time the BAR is accessed. The BAR remap optimization reduces this overhead by limiting the handling of BAR accesses. In the BAR remap optimization, GPUvm passes through the BAR accesses other than to GPU channel descriptors because GPUvm does not have to virtualize the values read from or written to the BAR areas except for GPU channel descriptors. Even if the BAR accesses are passed through, they must be isolated among multiple VMs. This isolation is achieved by making use of shadow page tables. The BAR accesses from the host CPU all go through GPU page tables; the offsets in the BAR areas are regarded as virtual addresses in GPU memory and translated to GPU physical addresses through the shadow page tables. By setting shadow page tables appropriately, all the accesses to the BAR areas are isolated among VMs.

Para-virtualization: Shadowing of GPU page tables is a major source of overhead in full-virtualization, because the entire page table needs to be scanned to detect changes to the guest GPU page tables. To reduce the cost of detecting the updates, we take a para-virtualization approach. In this approach, the guest GPU page tables are placed within the memory areas under the control of GPUvm and cannot be directly updated by guest GPU drivers. To update the guest GPU page tables, the guest GPU driver issues hypercalls to GPUvm. GPUvm validates the correctness of the page table updates when the hypercall is issued. This approach is inspired by the direct paging in Xen para-virtualization [3].

Multicall: Hypercalls are expensive because the context is switched to the hypervisor. To reduce the number of hypercalls, GPUvm provides a multicall interface that can batch several hypercalls into one. For example, instead of providing a hypercall that updates one page table entry at once, GPUvm provides a hypercall that allows multiple page table entries to be updated by one hypercall. The multicall is borrowed from Xen.

4 Implementation

Our prototype of GPUvm uses Xen 4.2.0, where both the domain 0 and the domain U adopt the Linux kernel

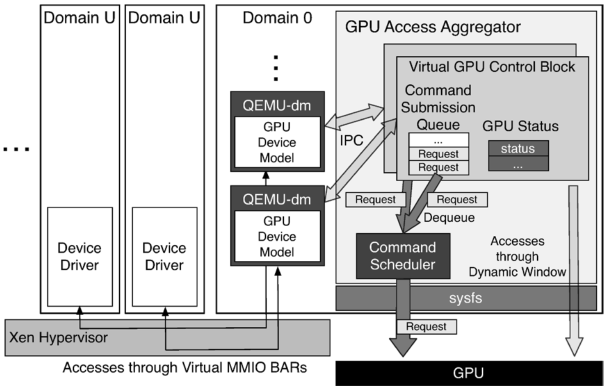


Figure 3: The prototype implementation of GPUvm.

v3.6.5. We target the device model of NVIDIA GPUs based on the Fermi/Kepler architectures [30, 31]. While full-virtualization does not require any modification to guest system software, we make a small modification to the GPU device driver called Nouveau, which is provided as part of the mainline Linux kernel, to implement our GPU para-virtualization approach.

Figure 3 shows the overview of implementation of the GPU Access Aggregator and its interactions with other components. The GPU device model is exposed in the domain U, which is created by the QEMU-dm and behaves as a virtual GPU. Guest device drivers in the domain U consider it as a normal GPU. It also exposes MMIO PCIe BARs, handling accesses to the BARs in Xen. All accesses to the GPU arbitrated by the GPU Access Aggregator are committed to the GPU through the `sysfs` interface.

The GPU device model communicates with the GPU Access Aggregator in the domain 0, using the POSIX inter-process communication (IPC). The GPU Access Aggregator is a user process in the domain 0, which receives requests from the GPU device model, and issues the aggregated requests to the physical GPU.

The GPU Access Aggregator has virtual GPU control blocks and the GPU command scheduler, which represent the state of virtual GPUs. The GPU device models update their own virtual GPU control blocks using IPC to manage the states of corresponding virtual GPUs when privileged events such as control register changes are issued from domain U.

Each virtual GPU control block maintains a queue to store command submission requests issued from the GPU device model. These command submission requests are scheduled to control GPU executions. This command scheduling mechanism is similar to TimeGraph [20]. However, the GPU command scheduler of GPUvm differs from TimeGraph in that it does not use GPU interrupts. It is very difficult, if not impossible, for the GPU Access Aggregator to insert the interrupt command to the original sequence of commands, because user contexts may also use some interrupt commands, and the GPU Access Aggregator cannot recognize them once they are fired. Therefore, our prototype implementation uses a thread-based scheduler polling on the request queue. Whenever command submission requests are stored in the queue, the scheduler dispatches them to the GPU. To calculate GPU time, our prototype polls a GPU control register value that is modified by the hardware just after GPU channels become active/inactive.

Another task of the GPU Access Aggregator is to manage GPU memory and maintain isolation of multiple VMs on partitioned memory resources. For this purpose, GPUvm creates shadow page tables and channel descriptors in the reserved area of GPU memory.

5 Experiments

To demonstrate the effectiveness of GPUvm, we conducted detailed experiments using a relevant commodity GPU. The objective of this section is to answer the following fundamental questions:

1. How much is the overhead of GPU virtualization incurred by GPUvm?
2. How does the number of GPU contexts affect performance?
3. Can multiple VMs meet fairness on GPU resources?

The experiments were conducted on a DELL PowerEdge T320 machine with eight Xeon E5-24700 2.3 GHz processors, 16 GB of memory, and two 500 GB SATA hard disks. We use NVIDIA Quadro 6000 for the target GPU, which is based on the NVIDIA Fermi architecture. We ran our modified Xen 4.2.0, assigning 4 GB and 1 GB of memory to the domain 0 and the domain U, respectively. In the domain U, Nouveau was running as the GPU device driver and Gdev [21] was running as the CUDA runtime. The following eight schemes were evaluated: **Native** (non-virtualized Linux 3.6.5), **PT** (pass-through provided by Xen's pass-through feature), **FV Naive** (full-virtualization w/o any optimization techniques), **FV BAR-Remap** (full-virtualization w/ BAR Remapping), **FV Lazy** (full-virtualization w/ Lazy Shadowing), **FV Optimized** (full-virtualization w/ BAR Remapping and Lazy Shadowing), **PV Naive** (para-virtualization w/o multicall), and **PV Multicall** (para-virtualization w/ multicall).

5.1 Overhead

To identify the overhead of GPU virtualization incurred by GPUvm, we run the well-known GPU benchmarks called Rodinia [5] as well as our microbenchmarks, as listed in Table 1. We measure their execution time on the eight platforms.

Table 1: List of the GPU benchmarks.

Benchmark	Description
NOP	No GPU operation
LOOP	Long-loop compute without data
MADD	1024x1024 matrix addition
MMUL	1024x1024 matrix multiplication
FMADD	1024x1024 matrix floating addition
FMMUL	1024x1024 matrix floating multiplication
CPY	64MB of HtoD and DtoH
PINCPY	CPY using pinned host I/O memory
BP	Back propagation (pattern recognition)
BFS	Breadth-first search (graph algorithm)
HS	Hotspot (physics simulation)
LUD	LU decomposition (linear algebra)
SRAD	Speckle reducing anisotropic diffusion (imaging)
SRAD2	SRAD with random pseudo-inputs (imaging)

5.1.1 Results

Figure 4 shows the execution time of the benchmarks on each platform. The x-axis lists the benchmark names while the y-axis exhibits the execution time normalized by one of Native. It is clearly observed that the overhead of GPU full-virtualization is mostly unacceptable, but our optimization techniques significantly contribute to reduction of this overhead. The execution times obtained in FV Naive are more than 100 times slower in nine benchmarks (nop, loop, madd, fmadd, fmmul, bp, hfs, hs, lid) than those obtained in Native. This overhead can be mitigated by using the BAR Remap and the Lazy Shadowing optimization techniques. Since these optimization techniques are complementary to each other, putting it together achieves more performance gain. The execution time is 6 times shorter in madd (the best case) while being 5 times shorter in mmul (the worst case). In some benchmarks, PT exhibits slightly faster performance than Native, especially in madd is 1.5 times shorter than PT. This is a GPU's mysterious behavior.

From these experimental results, we also find that GPU para-virtualization is much faster than full virtualization. The execution times obtained in PV Naive are 3 - 10 times slower than those obtained in Native except for pincpy. We discuss this reason in the next section. This overhead can also be reduced by our reduced hypercalls feature. The execution time increased in PV Multicall is at most 3 times.

5.1.2 Breakdown

The breakdown on the execution time of the GPU benchmarks is shown in Figure 5. We divide the total execution time into five phases; *init*, *htod*, *launch*, *dtoh*, and *close*. Init is time for setting up the GPU to execute a GPU kernel. Htod is time for host-to-device data transfers. Launch is time for the calculation on GPUs. Dtoh is device-to-host data transfer time. Close is time for destroying the GPU kernel. The figure indicates that the dominant factor of execution time in GPUvm is init and close phases. This tendency is significant for four GPUvm's full virtualization configurations. In FV Naive, init and close phases are more than 90 % in the execution times. By using optimization techniques, the phases' ratio becomes lowered.

Table 2 and 3 list BAR3 writes and shadow page table update counts in each benchmark. BAR3 is used for a memory aperture. BAR remapping achieves more performance gain in benchmarks that write more bytes in the BAR3 region. For example, the execution time of pincpy that writes 64 MB is 2 times shorter in FV BAR-Remap than FV Naive. Also, lazy shadowing works more effectively to the benchmarks that update shadow page tables more frequently. Specifically, the execution time of srad, where the shadow page table is updated in 52 times, is 2 times shorter in FV Lazy than FV Naive.

On the other hand, init and close phases in two GPUvm's para-virtualized platforms are much shorter than the full-virtualization configuration in all cases. Full-virtualization GPUvm performs many shadowing operations, including TLB flushes, since memory allocation are done frequently in the two phases. This cost can be significantly reduced in para-virtualization GPUvm in which memory allocations are requested by hypercall. Time spent in these phases is longer in pincpy since it issues more hypercalls than the other benchmarks. Table 4 lists the number of hypercall issues of each benchmark in PV Naive and PV Multicall. Compared to the other benchmarks, pincpy issues much more hypercalls. Also, our optimization dramatically reduces hypercall issues, resulting in the reduced overhead in PV Naive. As a result, the execution times in PV Multicall are close to those of PT and Native, compared with PV Naive's result. For example, the results in 7 benchmarks (mmul, cpu, pincpy, bp, hfs, srad, srad2) are similar in PV Multicall, PT, and Native.

5.2 Performance at Scale

To discern the overhead GPUvm incurs in multiple GPU contexts, we generate GPU workloads and measure their execution time in two scenarios; in the first scenario, one VM executes multiple GPU tasks, in the other scenario, multiple VM executes GPU tasks. We first launch 1, 2, 4, 8 GPU tasks in one VM with full-virtualized, para-virtualized, pass-throughed, GPU (*FV(1VM), PV(1VM), and PT*). These tasks are also run on native Linux (*Native*). Next, we prepare 1, 2, 4, 8 VMs and execute one GPU task on each VM with a full- or para-virtualized GPU (*FV and PV*) where all our optimizations are turned on. In each scenario, we run *madd* listed in Table 1. Specifically, we repeat the GPU kernel execution of madd 10000 times, and measure its execution time.

The results are shown in Fig. 6. The x-axis is the number of launched GPU contexts and the y-axis represents execution time. This figure reveals two points.

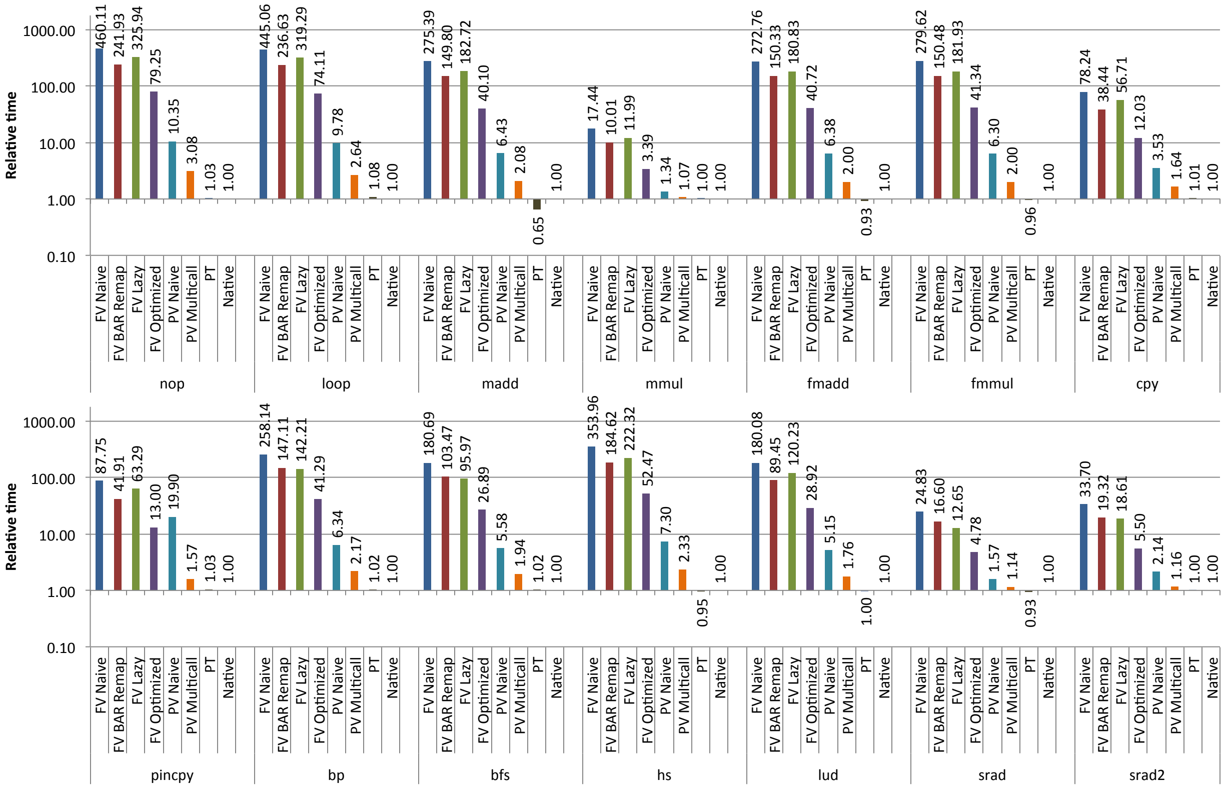


Figure 4: Execution time of the GPU benchmarks on the eight platforms.

Table 2: Total size of BAR access (bytes).

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
READ	0	0	0	0	0	0	0	0	0	0	0	0	0	0
WRITE	6688	6696	6648	6672	6680	6688	6168	268344	7280	6232	6736	7240	6352	6248

One is that full-virtualized GPUvm incurs larger overhead as GPU contexts are more. Time spent in init and close phases is longer in FV and FV (1VM) since GPUvm performs exclusive accesses to some GPU resources including the dynamic window. The other is that para-virtualized GPUvm achieves similar performance to pass-through GPU. The total execution time in PV and PV (1VM) is quite similar to those in PT even if GPU contexts are more. The kernel execution times in both FV (1VM) and FV are larger than the other GPU configurations, while those in the three GPU configurations are longer in 4 and 8 GPU contexts. This comes from GPUvm's overhead that it polls a GPU register to detect GPU kernel completion through MMIO.

5.3 Performance Isolation

To demonstrate how GPUvm achieves performance isolation among VMs, we launch a GPU workload on 2, 4, 8 VMs and measure each VM's GPU usage. For comparison, we use three schedulers; FIFO, CREDIT, and BAND scheduler. FIFO issues GPU requests in a first-in/first-out manner. CREDIT schedules GPU requests in a proportional fair share manner. Specifically, CREDIT reduces credits assigned to a VM in advance when its GPU requests are executed, and issues GPU requests of a VM whose credits are highest. BAND is our scheduler described in Sec.3.5. We prepare two GPU tasks; the one is madd, used in the previous experiment, and the other is an extended madd (*long-madd*), which performs 5 times more calculations than the regular madd. Each VM loops one of them. We run each task on a half of the VMs, respectively. For example, madd runs on 2 VMs while long-madd runs on 2 VMs in the 4-VM case.

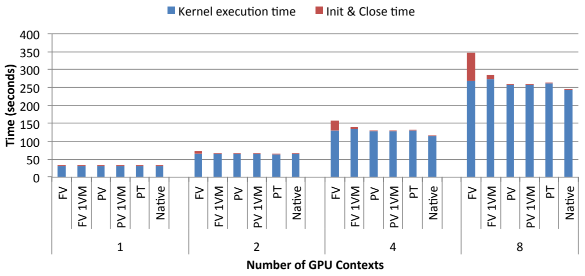


Figure 6: Performance across multiple VMs.

Fig. 7 shows the results. The x-axis represents the elapsed time and the y-axis is VM's GPU usage over 500 msec. The figure reveals that BAND is the only scheduler that achieves performance isolation in all cases. In

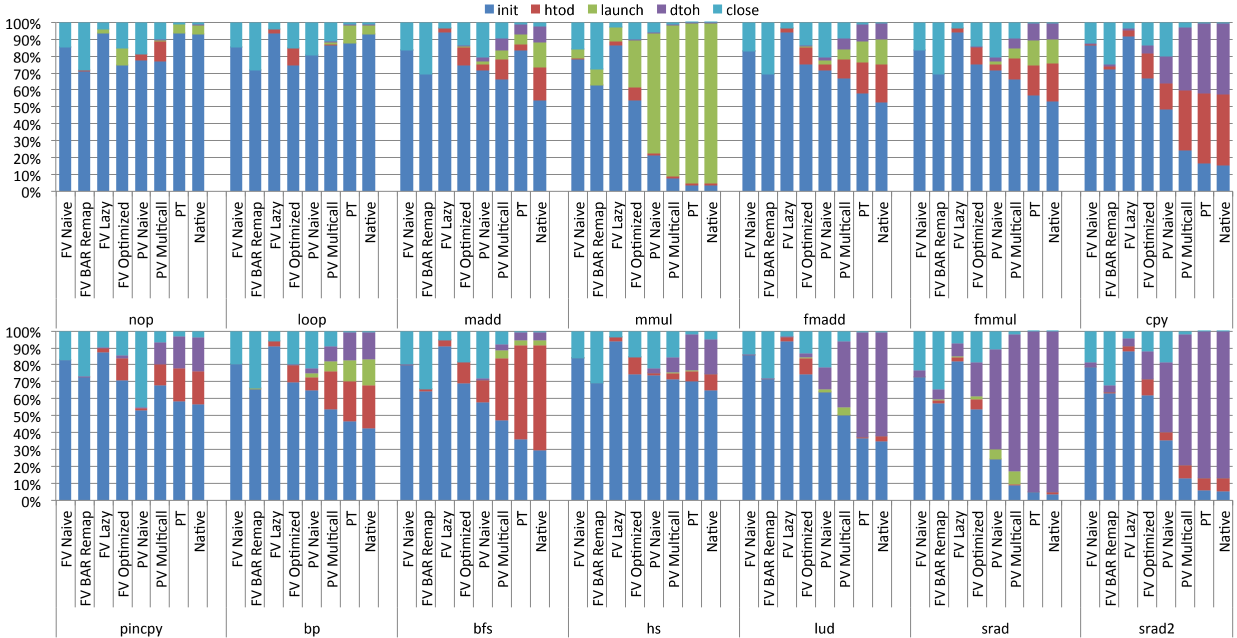


Figure 5: Breakdown on execution time of the GPU benchmarks.

Table 3: Update count for GPU shadow page tables.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
FV Naive	30	30	34	34	34	34	26	28	40	42	34	30	52	40
FV Optimized	7	7	7	7	7	7	6	6	7	7	7	7	7	7

FIFO, GPU usages in long-madd are higher in all cases since FIFO dispatches more GPU commands from long-madd than madd. CREDIT fails to achieve the fairness among VMs in 2-VM case. Since the command submission request queue contains the requests only from long-madd just after the madd's commands completed, CREDIT dispatches the long-madd's requests. As a result, the VM running madd has to wait for the completion of the long-madd's commands. BAND waits for request arrivals for a short period just after a GPU kernel completes. BAND can handle the requests issued from the VMs whose GPU usage is less.

CREDIT achieves fair-share GPU scheduling in 4- and 8- VM. In these cases, CREDIT has more opportunities to dispatch less-credit VMs' commands for the following two reasons. First, GPUvm's queue has GPU command submission requests from two or more VMs just after a GPU kernel completes, differently from the 2-VM case. Second, GPUvm submits GPU operations from three or more VMs that complete shortly; GPUvm can have more scheduling points.

Note that BAND cannot achieve fairness among VMs in a fine-grained manner on the current GPUs. Fig. 8 shows VMs' GPU usages over 100 msec. Even with BAND, the GPU usages are fluctuated over time. This is because the GPU is a non-preemptive device. To achieve finer-grained GPU fair-share scheduling, we need a novel mechanism inside the GPU which effectively switches GPU kernels.

6 Related Work

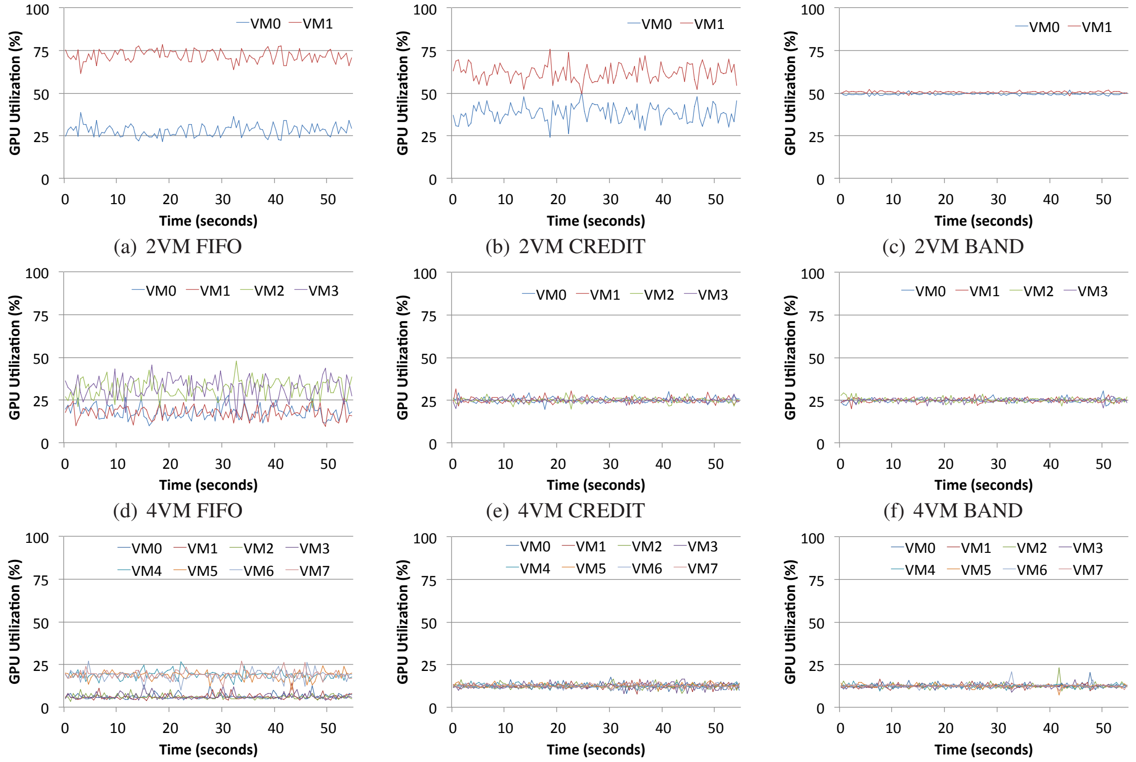
Table 5 briefly summarizes the characteristics of several GPU virtualization schemes, and compares them to GPUvm. Some vendors invent techniques of GPU virtualization. NVIDIA has announced NVIDIA VGX [32], which exploits virtualization supports of Kepler generation GPUs. These are proprietary so their details are closed. To the best of our knowledge, GPUvm is the first open architecture of GPU virtualization offered by the hypervisor. GPUvm carefully selects resources to virtualize, GPU page tables and channels, to keep its applicability to various GPU architectures.

VMware SVGA2 [7] para-virtualizes GPUs to mitigate the overhead of virtualizing GPU graphics features. The SVGA2 handles graphics-related requests by using an architecture-independent communication to efficiently perform 3D rendering and hide GPU hardware. While this approach is specific to graphics acceleration, GPUvm coordinates interactions between GPUs and guest device drivers.

Gottschalk et al. proposes low-overhead GPU virtualization, named LoGV, for GPGPU applications [10]. Their approach is categorized into para-virtualization where device drivers in VMs send requests for resource allocation and mapping memory into system RAM to the hypervisor. Similar to our work, this work exhibits para-virtualization mechanisms to minimize GPGPU virtualization overhead. Our work reveals which virtualization technique for GPUs is efficient in a quantitative way.

Table 4: The number of hypercall issues.

	nop	loop	madd	mmul	fmadd	fmmul	cpy	pincpy	bp	bfs	hs	lud	srad	srad2
PV Naive	1230	1230	1420	1420	1420	1420	2010	34784	1628	1993	1169	1429	1985	2681
PV Multicall	93	93	97	97	97	97	81	218	117	117	97	89	149	107



(a) 2VM FIFO (b) 2VM CREDIT (c) 2VM BAND

(d) 4VM FIFO (e) 4VM CREDIT (f) 4VM BAND

(g) 8VM FIFO (h) 8VM CREDIT (i) 8VM BAND

Figure 7: VMs' GPU usages (over 500 ms) on the three GPU schedulers.

API remoting, in which API calls are forwarded from the guest to the host which has the GPU, have been studied widely. GViM [11], vCUDA [37], and rCUDA [8] forward CUDA APIs. VMGL [24] achieves API remoting of OpenGL. gVirtuS [9] supports API remoting of CUDA, OpenCL, a part of OpenGL. In these approaches, applications are inherently limited to APIs the wrapper-libraries offer. Keeping the wrapper-libraries compatible to the original ones is not trivial task since new functionalities are frequently integrated into GPU libraries including CUDA and OpenCL. Moreover API remoting requires that the whole GPU software stacks including device drivers and runtimes become part of the TCB.

Amazon EC2 [1] provides GPU instances. It makes use of pass-through technology to expose a GPU to an instance. Since a pass-throughed GPU is directly managed by the guest OS, we cannot multiplex the GPU on a physical host.

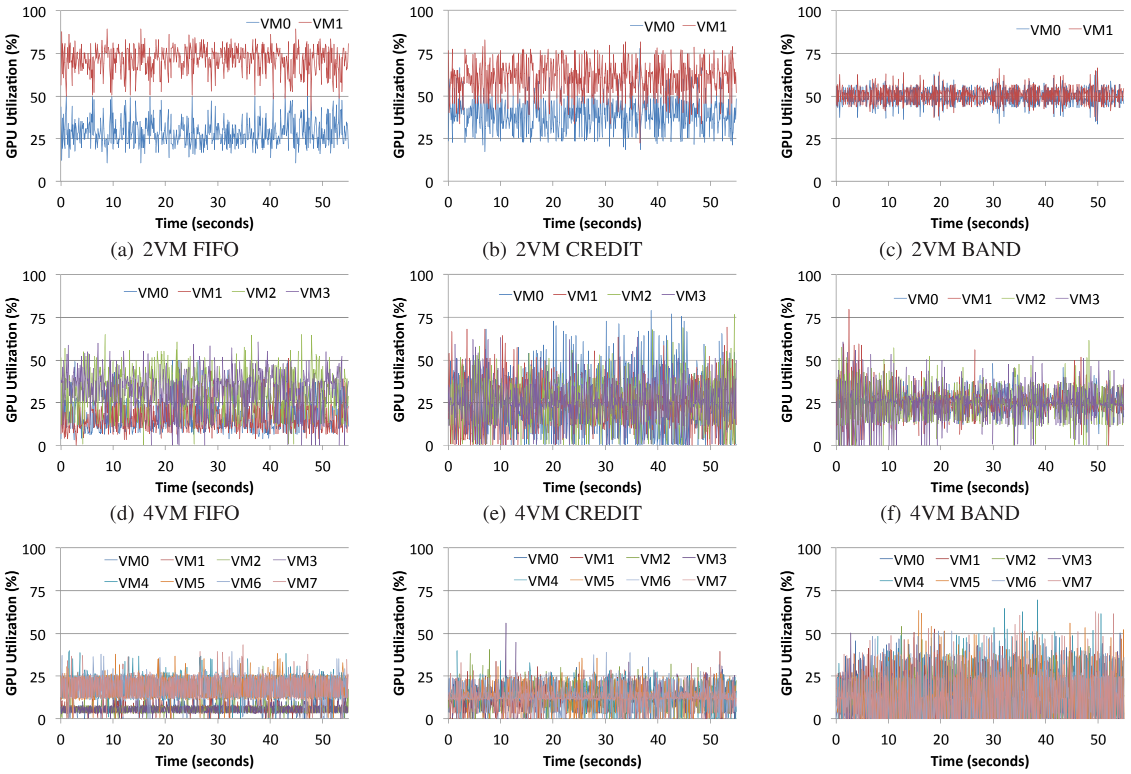
GPUvm is complementary to GPU command scheduling methods. VGRIS [41] enables us to schedule GPU commands in SLA-aware, proportional-share or the hybrid scheduling. Pegasus [12] coordinates GPU command queuing and CPU dispatching so that multi-VMs can effectively share CPU and GPU resources. Disengaged Scheduling [29] applies fair queuing scheduling with a probabilistic extension to GPU, and provides protection and fairness without compromising efficiency.

XenGT [16] is GPU virtualization for Intel on-chip GPUs with the similar design to GPUvm. Our work provides detailed analysis of the performance bottlenecks of the current GPU virtualization. It is useful to design the architecutre of the GPU virtualization, and also useful for GPU vendors to design the future GPU architecture which supports virtualization.

Some work aims at the efficient management of GPUs at the operating system layer such as GPU command scheduler [20], kernel-level GPU runtime [21], OS abstraction of GPUs [34,35] and file system for GPUs [39]. These mechanisms can be incorporated into GPUvm.

7 Conclusion

In this work, we try to answer the question; why not virtualizing GPU at the hypervisor? This paper have presented GPUvm, an open architecture of GPU virtualization. GPUvm supports full-virtualization and para-virtualization with optimization techniques. Experimental results using our prototype showed that full-



(a) 2VM FIFO (b) 2VM CREDIT (c) 2VM BAND

(d) 4VM FIFO (e) 4VM CREDIT (f) 4VM BAND

(g) 8VM FIFO (h) 8VM CREDIT (i) 8VM BAND

Figure 8: VMs' GPU usages (over 100 ms) on the three GPU schedulers.

Table 5: A comparison of GPUvm to other GPU virtualization schemes.

	Category	W/o Library mod.	W/o kernel mod.	Multiplexing	Scheduling
vCUDA [37], rCUDA [8]	API Remoting	×	√	√	×
VMGL [24]		×	√	√	×
VGRIS(gVirtuS) [9,41]		×	√	√	SLA-aware, Proportional-share
Pegasus(GViM) [11,12]		×	√	√	SLA-aware, Throughput-based, Proportional fair-share
SVGA2 [7]	Para-Virt.	√	×	√	×
LoGV [10]		√	×	√	×
GPU Instances [1]	Pass-through	√	×	×	×
GPUvm	Full-Virt.& Para-Virt.	√	√	√	Credit-based fair-share

virtualization exhibits non-trivial overhead largely due to MMIO handling, and para-virtualization provides yet two or three times slower performance than pass-through and native approaches. Also the results reveal that para-virtualization is preferred in performance, though highly compute-intensive GPU applications may also benefit from full-virtualization if their execution times are much larger than the overhead of full-virtualization.

For future directions, it should be investigated that the optimization techniques proposed in vIOMMU [2] can be applied to GPUvm. We hope our experience in GPUvm gives insight into designing the support of device virtualization such as SR-IOV [6].

8 Acknowledgements

This work was supported in part by the Grant-in-Aid for Scientific Research B (25280043) and C (23500050).

References

[1] Amazon.com. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`, 2014.

[2] Amit, N., Ben-Yehuda, M., Tsafrir, D., and Schuster, A. vIOMMU: Efficient IOMMU Emulation. In *USENIX ATC* (2011), pp. 73–86.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the Art of Virtualization. In *ACM SOSP* (2003), pp. 164–177.

[4] Basaran, C., and Kang, K.-D. Supporting Preemptive Task Executions and Memory Copies in GPGPUs. In *Proc. of Euromicro Conf. on Real-Time Systems* (2012), IEEE, pp. 287–296.

[5] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Proc. of IEEE Int'l Symp. on Workload Characterization* (2009), pp. 44–54.

[6] Dong, Y., Yu, Z., and Rose, G. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Proc. of Workshop on I/O Virtualization* (2008).

[7] DOWTY, M., AND SUGERMAN, J. GPU Virtualization on VMware's Hosted I/O Architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[8] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTI, E. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *Proc. of IEEE Int'l Conf. on High Performance Computing Simulation* (2010), pp. 224–231.

[9] GIUNTA, G., MONTELLA, R., AGRILLO, G., AND COVIELLO, G. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds. In *Proc. of Int'l Euro-Par Conf. on Parallel Processing* (2010), Springer, pp. 379–391.

[10] GOTTSCHLAG, M., HILLENBRAND, M., KEHNE, J., STOESS, J., AND BELLOSA, F. LoGV: Low-overhead GPGPU Virtualization. In *Proc. of Int'l Workshop on Frontiers of Heterogeneous Computing* (2013), IEEE, pp. 1721–1726.

[11] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. GViM: GPU-Accelerated Virtual Machines. In *Proc. of ACM Workshop on System-level Virtualization for High Performance Computing* (2009), pp. 17–24.

[12] GUPTA, V., SCHWAN, K., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In *USENIX ATC* (2011), pp. 31–44.

[13] HAN, S., JANG, K., PARK, K., AND MOON, S. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review 40*, 4 (2010), 195–206.

[14] HE, B., YANG, K., FANG, R., LU, M., GOVINDARAJU, N., LUO, Q., AND SANDER, P. Relational Joins on Graphics Processors. In *ACM SIGMOD* (2008), pp. 511–524.

[15] HIRABAYASHI, M., KATO, S., EDAHIRO, M., TAKEDA, K., KAWANO, T., , AND MITA, S. Gpu implementations of object detection using hog features and deformable models. In *Proc. of IEEE Int'l Conf. on Cyber-Physical Systems, Networks, and Applications* (2013), pp. 106–111.

[16] INTEL. Xen - Graphics Virtualization (XenGT). `https://01.org/xen/blogs/srclarkx/2013/graphics-virtualization-xengt`, 2013.

[17] JANG, K., HAN, S., HAN, S., MOON, S., AND PARK, K. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *USENIX NSDI* (2011), pp. 1–14.

[18] KALDEWEY, T., LOHMAN, G. M., MÜLLER, R., AND VOLK, P. B. GPU Join Processing Revisited. In *Proc. of Int'l Workshop on Data Management on New Hardware* (2012), pp. 55–62.

[19] KATO, S., AUMILLER, J., AND BRANDT, S. Zero-copy I/O processing for low-latency GPU computing. In *Proc. of ACM/IEEE Int'l Conf. on Cyber-Physical Systems* (2013), pp. 170–178.

[20] KATO, S., LAKSHMANAN, K., RAJKUMAR, R., AND ISHIKAWA, Y. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *USENIX ATC* (2011), pp. 17–30.

[21] KATO, S., MCTHROW, M., MALTZAHN, C., AND BRANDT., S. Gdev: First-Class GPU Resource Management in the Operating System. In *USENIX ATC* (2012), pp. 401–412.

[22] KIM, C., CHHUGANI, J., SATISH, N., SEDLAR, E., NGUYEN, A. D., KALDEWEY, T., LEE, V. W., BRANDT, S. A., AND DUBEY, P. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *ACM SIGMOD* (2010), pp. 339–350.

[23] KOSCIELNICKI, M. Envytools. `https://0x04.net/envytools.git`, 2014.

[24] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. VMM-Independent Graphics Acceleration. In *ACM VEE* (2007), pp. 33–43.

[25] LINUX OPEN-SOURCE COMMUNITY. Nouveau Open-Source GPU Device Driver. `http://nouveau.freedesktop.org/`, 2014.

[26] MARUYAMA, N., NOMURA, T., SATO, K., AND MATSUOKA, S. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–12.

[27] MCNAUGHTON, M., URMSON, C., DOLAN, J. M., AND LEE, J.-W. Motion Planning for Autonomous Driving with a Conformal Spatiotemporal Lattice. In *Proc. of IEEE Int'l Conf. on Robotics and Automation* (2011), pp. 4889–4895.

[28] MENYCHTAS, K., SHEN, K., AND SCOTT, M. L. Enabling OS Research by Inferring Interactions in the Black-Box GPU Stack. In *USENIX ATC* (2013), pp. 291–296.

[29] MENYCHTAS, K., SHEN, K., AND SCOTT, M. L. Disengaged scheduling for fair, protected access to fast computational accelerators. In *ACM ASPLOS* (2014), pp. 301–316.

[30] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. `http://www.nvidia.com/`, 2009.

[31] NVIDIA. NVIDIA's next generation CUDA computer architecture: Kepler GK110. `http://www.nvidia.com/`, 2012.

[32] NVIDIA. NVIDIA GRID VGX SOFTWARE. `http://www.nvidia.com/object/grid-vgx-software.html`, 2014.

[33] RATH, N., BIALEK, J., BYRNE, P., DEBONO, B., LEVESQUE, J., LI, B., MAUEL, M., MAURER, D., NAVRATIL, G., AND SHIRAKI, D. High-speed, multi-input, multi-output control using GPU processing in the HBT-EP tokamak. *Fusion Engineering and Design* (2012), 1895–1899.

[34] ROSSBACH, C. J., CURREY, J., SILBERSTEIN, M., RAY, B., AND WITCHEL, E. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *ACM SOSP* (2011), pp. 223–248.

[35] ROSSBACH, C. J., YU, Y., CURREY, J., MARTIN, J.-P., AND FETTERLY, D. Dandelion: a Compiler and Runtime for Heterogeneous Systems. In *ACM SOSP* (2013), pp. 49–68.

[36] SATISH, N., KIM, C., CHHUGANI, J., NGUYEN, A. D., LEE, V. W., KIM, D., AND DUBEY, P. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *ACM SIGMOD* (2010), pp. 351–362.

[37] SHI, L., CHEN, H., SUN, J., AND LI, K. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Transactions on Computers 61*, 6 (2012), 804–816.

[38] SHIMOKAWABE, T., AOKI, T., TAKAKI, T., ENDO, T., YAMANAKA, A., MARUYAMA, N., NUKADA, A., AND MATSUOKA, S. Peta-scale Phase-Field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer. In *Proc. of Int'l Conf. for High Performance Computing, Networking, Storage and Analysis* (2011), pp. 1–11.

[39] SILBERSTEIN, M., FORD, B., KEIDAR, I., AND WITCHEL, E. GPUfs: Integrating a File System with GPUs. In *ACM ASPLOS* (2013), pp. 485–498.

[40] SUN, W., RICCI, R., AND CURRY, M. L. GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel. In *Proc. of Int'l Systems and Storage Conf.* (2012), pp. 9:1–9:12.

[41] YU, M., ZHANG, C., QI, Z., YAO, J., WANG, Y., AND GUAN, H. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming. In *ACM HPDC* (2013), pp. 203–214.

A Full GPU Virtualization Solution with Mediated Pass-Through

Kun Tian, Yaozu Dong, David Cowperthwaite
Intel Corporation

Abstract

Graphics Processing Unit (GPU) virtualization is an enabling technology in emerging virtualization scenarios. Unfortunately, existing GPU virtualization approaches are still suboptimal in performance and full feature support.

This paper introduces gVirt, a product level GPU virtualization implementation with: 1) *full GPU virtualization* running native graphics driver in guest, and 2) *mediated pass-through* that achieves both good performance and scalability, and also secure isolation among guests. gVirt presents a virtual full-fledged GPU to each VM. VMs can directly access performance-critical resources, without intervention from the hypervisor in most cases, while privileged operations from guest are trap-and-emulated at minimal cost. Experiments demonstrate that gVirt can achieve up to 95% native performance for GPU intensive workloads, and scale well up to 7 VMs.

1. Introduction

The Graphics Processing Unit (GPU) was originally invented to accelerate graphics computing, such as gaming and video playback. Later on, GPUs were used in high performance computing, as well, such as image processing, weather broadcast, and computer aided design. Currently, GPUs are also commonly used in many general purpose applications, with the evolution of modern windowing systems, middleware, and web technologies.

As a result, rich GPU applications present rising demand for *full GPU virtualization* with good performance, full features, and sharing capability. Modern desktop virtualization, either locally on clients such as XenClient [35] or remotely on servers such as VMware Horizon [34], requires GPU virtualization to support uncompromised native graphical user experience in a VM. In the meantime, cloud service providers start to build GPU-accelerated virtual instances, and sell GPU computing resources as a service [2]. Only *full GPU virtualization* can meet the diverse requirements in those usages.

However, there remains the challenge to implement *full GPU virtualization*, with a good balance among performance, features and sharing capability. Figure 1 shows the spectrum of GPU virtualization solutions (with hardware acceleration increasing from left to right). *Device emulation* [7] has great complexity and extremely low performance, so it does not meet today's needs. *API forwarding* [3][9][22][31] employs a frontend driver, to forward the high level API calls inside a VM, to the host for acceleration. However, API forwarding faces the challenge of supporting full features, due to the complexity of intrusive modification in the guest graphics software stack, and incompatibility between the guest and host graphics software stacks. *Direct pass-through* [5][37] dedicates the GPU to a single VM, providing full features and the best performance, but at the cost of device sharing capability among VMs. *Mediated pass-through* [19], passes through performance-critical resources, while mediating privileged operations on the device, with good performance, full features, and sharing capability.

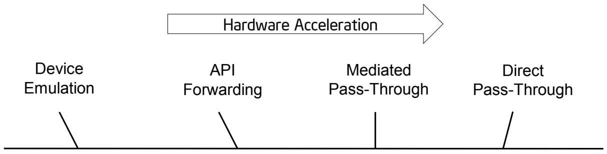


Figure 1: The spectrum of I/O virtualization

This paper introduces gVirt, the first product level GPU virtualization implementation, to our knowledge, with: 1) *full GPU virtualization* running a native graphics driver in guest, and 2) *mediated pass-through* that achieves good performance, scalability, and also secure isolation among guests. A virtual GPU (vGPU), with full GPU features, is presented to each VM. VMs can directly access performance-critical resources, without intervention from the hypervisor in most cases, while privileged operations from guest are trap-and-emulated to provide secure isolation among VMs. The vGPU context is switched per quantum, to share the physical GPU among multiple VMs without user notice. As such, gVirt achieves *full GPU virtualization*, with a great balance among performance, features, and sharing capability. We implement gVirt in Xen, with integrated Intel® Processor Graphics [13] in the 4th generation Intel® Core™ processor. The principles and architecture of gVirt, however, is also applicable to different GPUs and hypervisors. gVirt was initially presented at the Xen Summit [10], and all the gVirt source code is now available to the open source community [8].

This paper overcomes a variety of technical challenges and makes these contributions:

- Introduces a *full GPU virtualization* solution with mediated pass-through that runs the native graphics driver in guest
- Passes through performance-critical resource accesses with graphics memory resource partitioning, address space ballooning, and direct execution of guest command buffer
- Isolates guests by auditing and protecting the command buffer at the time of command submission, with smart shadowing
- Further improves performance with virtualization extension to the hardware specification and the graphics driver (less than 100 LOC changes to the Linux kernel mode graphics driver)
- Provides a product level open source code base for follow-up research on GPU virtualization, and a comprehensive evaluation for both Linux and Windows guests
- Demonstrates that gVirt can achieve up to 95% of native performance for GPU-intensive workloads, and up to 83% for workloads that stress both the CPU and GPU

The rest of the paper is organized as follows. An overview of the GPU is provided in section 2. In section 3, we present the design and implementation of gVirt. gVirt is evaluated with a combination of graphics workloads, in section 4. Related work is discussed in section 5, and future work and conclusion are in section 6.

2. GPU Programming Model

In general, Intel Processor Graphics works as shown in Figure 2. The *render engine* fetches GPU commands from the command buffer, to accelerate rendering graphics in many different features. The *display engine* fetches pixel data from the frame buffer and then sends them to external monitors for display.

This architecture abstraction applies to most modern GPUs but may differ in how graphics memory is implemented. Intel Processor Graphics uses system memory as graphics memory, while other GPUs may use on-die memory. System memory can be mapped into multiple virtual address spaces by GPU page tables. A 2GB global virtual address space, called *global graphics memory,* accessible from both the GPU and CPU, is mapped through *global page table. Local graphics memory* spaces are supported in the form of multiple 2GB local virtual address spaces, but are only limited to access from the render engine, through *local page tables*. *Global graphics memory* is mostly the frame buffer, but also serves as the command buffer. Massive data accesses are made to *local graphics memory* when hardware acceleration is in progress. Other GPUs have some similar page table mechanism accompanying the on-die memory.

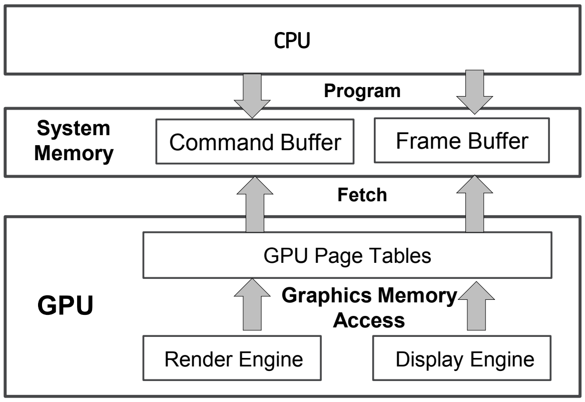


Figure 2: The architecture of the Intel Processor Graphics

The CPU programs the GPU through GPU-specific commands, shown in Figure 2, in a producer-consumer model. The graphics driver programs GPU commands into the *command buffer*, including primary buffer and batch buffer, according to high level programming APIs like OpenGL and DirectX. Then the GPU fetches and executes the commands. The primary buffer, a ring buffer (*ring buffer*), may chain other batch buffers (*batch buffer*) together. We use the terms: primary buffer and ring buffer, interchangeably hereafter. The batch buffer is used to convey the majority of the commands (up to ~98%) per programming model. A register tuple (*head, tail*) is used to control the ring buffer. The CPU submits the commands to the GPU by updating *tail*, while the GPU fetches commands from *head,* and then notifies the CPU by updating *head*, after the commands have finished execution.

Having introduced the GPU architecture abstraction, it is important for us to understand how real-world graphics applications use the GPU hardware so that we can virtualize it in VMs efficiently. To do so, we characterized, for some representative GPU-intensive 3D workloads (Phoronix Test Suite [28]), the usages of the four critical interfaces: the *frame buffer*, the *command buffer*, the GPU *Page Table Entries* (PTEs) which carry the GPU page tables, and the *I/O registers* including Memory-Mapped I/O (MMIO) registers, Port I/O (PIO) registers, and PCI configuration space registers for internal state. Figure 3 shows the average access frequency of running Phoronix 3D workloads on four interfaces.

The *frame buffer* and *command buffer* exhibit the most performance-critical resources, as shown in Figure 3. The detail test configuration is shown in section 4. When the applications are being loaded, lots of source vertexes and pixels are written by the CPU, so the frame buffer accesses dominate, in the 100s of thousands per second. Then at run-time, the CPU programs the GPU, through the commands, to render the frame buffer, so the command buffer accesses become the largest group, also in the 100s of thousands per second. PTE and I/O accesses are minor, in tens of thousands per second, in both load and run-time phases.

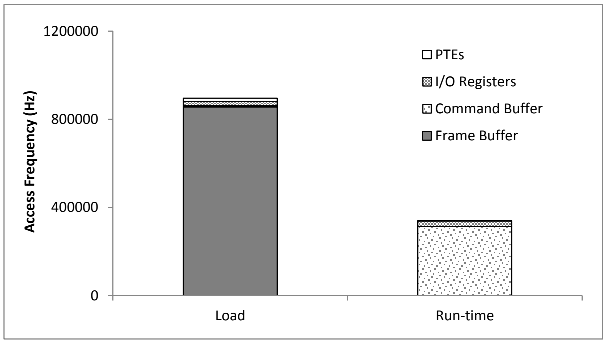


Figure 3: Access patterns of running 3D workloads

3. Design and Implementation

gVirt is a *full GPU virtualization* solution with mediated pass-through. As such, gVirt presents every VM a full-fledged GPU, to run native graphics driver inside a VM. The challenge, however, is significant in three ways: 1) complexity in virtualizing an entire sophisticated modern GPU, 2) performance due to multiple VMs sharing the GPU, and 3) secure isolation among the VMs without any compromise. gVirt reduces the complexity and achieves good performance, through the mediated pass-through technique, in subsection 3.1, 3.2, 3.3, and 3.4, and enforces the secure isolation, with the smart shadowing scheme in subsection 3.5.

3.1. Architecture

Figure 4 shows the overall architecture of gVirt, based on Xen hypervisor, with Dom0 as the privileged VM, and multiple user VMs. A gVirt stub module, in Xen hypervisor, extends the memory virtualization module (vMMU), including EPT for user VMs and PVMMU for Dom0, to implement the policies of trap and pass-through. Each VM runs the native graphics driver, and can directly access the performance-critical resources: the frame buffer and command buffer, with resource partitioning as presented in subsection 3.3 & 3.4. To protect privileged resources, that is, the I/O registers and PTEs, corresponding accesses, from the graphics driver in user VMs and Dom0, are trapped and forwarded to the mediator driver in Dom0 for emulation. The mediator uses hypercall to access the physical GPU. In addition, the mediator implements a GPU scheduler, which runs concurrently with the CPU scheduler in Xen, to share the physical GPU amongst VMs.

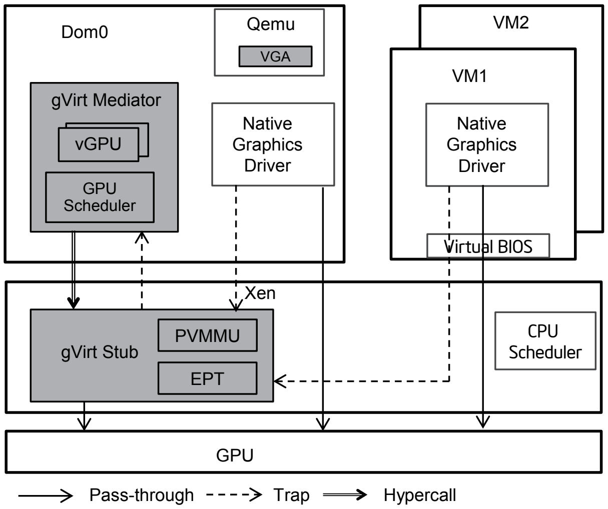


Figure 4: The gVirt Architecture

gVirt uses the physical GPU to directly execute all the commands submitted from a VM, so it avoids the complexity of emulating the render engine, which is the most complex part within the GPU. In the meantime, the resource pass-through, of both the frame buffer and command buffer, minimizes the hypervisor's intervention on the CPU accesses, while the GPU scheduler guarantees every VM a quantum for direct GPU execution. So gVirt achieves good performance when sharing the GPU amongst multiple VMs.

gVirt stub: We extend the Xen vMMU module, to selectively trap or pass-through guest access of certain GPU resources. Traditional Xen supports only pass-through or trap of the entire I/O resource of a device, for either device emulation or device pass-through. gVirt manipulates the EPT entries to selectively present or hide a specific address range to user VMs, while uses a reserved bit of PTEs in PVMMU for Dom0, to selectively trap or pass-through guest accesses to a specific address range. In both cases, the PIO accesses are trapped. All the trapped accesses are forwarded to the mediator for emulation, and the mediator uses hypercalls to access the physical GPU.

Mediator: gVirt mediator driver emulates virtual GPUs (vGPUs) for privileged resource accesses, and conducts context switches amongst the vGPUs. In the meantime, gVirt relies on the Dom0 graphics driver to initialize the physical device and to manage power. gVirt takes a flexible release model, by implementing the mediator as a kernel module in Dom0, to ease the binding between the mediator and hypervisor.

A split CPU/GPU scheduling mechanism is implemented in gVirt, for two reasons. First, the cost of the GPU context switch is over 1000X the cost of the CPU context switch (~700us vs. ~300ns, per our experiments). Second, the number of the CPU cores likely differs from the number of the GPU cores, in a computer system. So, gVirt implements a separate GPU scheduler from the existing CPU scheduler. The split scheduling mechanism leads to the requirement of concurrent accesses to the resources from both the CPU and the GPU. For example, while the CPU is accessing the graphics memory of VM1, the GPU may be accessing the graphics memory of VM2, concurrently.

Native driver: gVirt runs the native graphics driver inside a VM, which directly accesses a portion of the performance-critical resources, with privileged operations emulated by the mediator. The split scheduling mechanism leads to the resource partitioning design in subsection 3.3. To support resource partitioning better, gVirt reserves a Memory-Mapped I/O (MMIO) register window, called gVirt_info, to convey the resource partitioning information to the VM. Note that the location and definition of gVirt_info has been pushed to the hardware specification as a virtualization extension, so the graphics driver must handle the extension natively, and future GPU generations must follow the specification for backward compatibility. The modification is very limited, with less than 100 LOC changes to Linux kernel mode graphics driver.

Qemu: We reuse Qemu [7] to emulate the legacy VGA mode, with the virtual BIOS to boot user VMs. This design simplifies the mediator logic, because the modern graphics driver doesn't rely on the BIOS boot state. It re-initializes the GPU from scratch. The gVirt extension module decides whether an emulation request should be routed to the mediator or to Qemu.

3.2. GPU Sharing

The mediator manages vGPUs of all VMs, by trap-and-emulating the privileged operations. The mediator handles the physical GPU interrupt, and may generate virtual interrupt to the designated VMs. For example, a physical completion interrupt of command execution may trigger a virtual completion interrupt, delivered to the rendering owner. The idea of emulating a vGPU instance per semantics is simple; however, the implementation involves a large engineering effort and a deep understanding of the GPU. For example, ~700 I/O registers are accessed by the Linux graphics driver.

Render engine scheduling: gVirt scheduler implements a coarse-grain quality of service (QoS) policy. A time quantum of 16ms is selected as the scheduling time slice, because it is less human perceptible to image change. Such a relatively large quantum also comes from that, the cost of the GPU context switch is over 1000X that of the CPU context switch, so it can't be as small as the time slice in CPU scheduler. The commands from a VM are submitted to the GPU continuously, until the guest runs out of its time-slice. gVirt needs to wait for the guest ring buffer to become idle before switching, because most GPUs today are non-preemptive, which may impact the fairness. To minimize the wait overhead, gVirt implements a coarse-grain flow control mechanism, by tracking the command submission to guarantee the piled commands, at any time, are within a certain limit. Therefore, the time drift between the allocated time slice and the executed time is relatively small, compared to the large quantum, so a coarse-grain QoS policy is achieved.

Render context switch: gVirt saves and restores internal pipeline state and I/O register states, plus cache/TLB flush, when switching the render engine among vGPUs. The internal pipeline state is invisible to the CPU, but can be saved and restored through GPU commands. Saving/restoring I/O register states can be achieved through reads/writes to a list of the registers in the render context. Internal cache and Translation Lookaside Table (TLB), included in modern GPUs to accelerate data accesses and address translations, must be flushed using commands at render context switch, to guarantee isolation and correctness. The steps used to switch a context in gVirt are: 1) save current I/O states, 2) flush the current context, 3) use the additional commands to save the current context, 4) use the additional commands to restore the new context, and 5) restore I/O state of the new context.

gVirt uses a *dedicated ring buffer* to carry the additional GPU commands. gVirt may reuse the (audited) guest ring buffer for performance, but it is not safe to directly insert the commands into the guest ring buffer, because the CPU may continue to queue more commands as well, leading to overwritten content. To avoid the race condition, gVirt switches from the guest ring buffer to its own *dedicated ring buffer*. At the end of the context switch, gVirt switches from the *dedicated ring buffer* to the guest ring buffer of the new VM.

Display management: gVirt reuses the Dom0 graphics driver to initialize the display engine, and then manages the display engine to show different VM frame buffers. When two vGPUs have the same resolution, only the frame buffer locations are switched. For different resolutions, gVirt uses the hardware scalar, a common feature in modern GPUs, to scale the resolution up and down automatically. Both methods take mere milliseconds. In many cases, gVirt may not require the display management, if the VM is not shown on the physical display, for example, when it is hosted on the remote servers [34].

3.3. Pass-Through

gVirt passes through the accesses to the *frame buffer* and *command buffer* to accelerate performance-critical operations from a VM. For the *global graphics memory* space, 2GB in size, we propose graphics memory resource partitioning and address space ballooning mechanism. For the *local graphics memory* spaces, each with a size of 2GB too, we implement per-VM *local graphics memory*, through render context switch, due to *local graphics memory* only accessible by GPU.

Graphics memory resource partitioning: gVirt partitions the *global graphics memory* among VMs. As explained in subsection 3.1, split CPU/GPU scheduling mechanism requires that the *global graphics memory* of different VMs can be accessed simultaneously by the CPU and the GPU, so gVirt must, at any time, present each VM with its own resource, leading to the resource partitioning approaching, for *global graphics memory*, as shown in Figure 5.

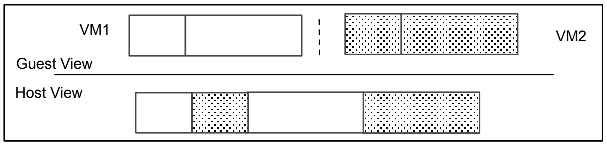


Figure 5: Graphics memory with resource partitioning

The performance impact of the reduced *global graphics memory* resource, due to the partitioning, is very limited according to our experiments. Results are shown in Figure 6, with performance normalized to the score of the default 2GB case. We did experiments in the native environment, and then scaled the 2GB *global graphics* memory down to 1/2, 1/4, and 1/8, with negligible performance impact observed. This is because the driver uses the *local graphics memory* to hold the massive rendering data, while the *global graphics memory* mostly serves only for the frame buffer, and the ring buffer, which are limited in size.

The resource partitioning also reveals an interesting problem: the guest and host now have an inconsistent view of the *global graphics memory*. The guest graphics driver is unaware of the partitioning, assuming with exclusive ownership: the *global graphics memory* is contiguous, starting from address zero. gVirt has to translate between the host view and the guest view, for any graphics address, before being accessed by the CPU and GPU. It therefore incurs more complexity and additional overhead, such as additional accesses to the command buffer (usually mapped as un-cacheable and thus slow on access).

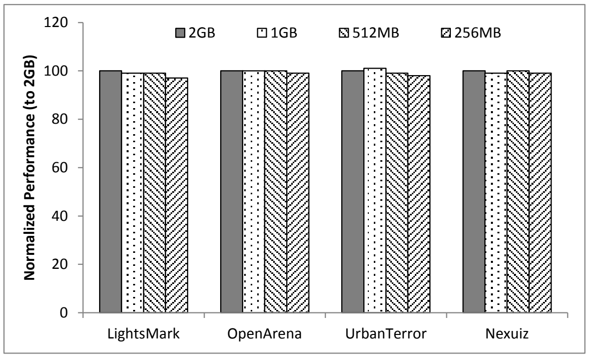


Figure 6: The performance with different size of the *global graphics memory*

Address space ballooning: We introduce the *address space ballooning* technique, to eliminate the address translation overhead, illustrated in Figure 7. gVirt exposes the partitioning information to the VM graphics driver, through the gVirt_info MMIO window. The graphics driver marks other VMs' regions as 'ballooned', and reserves them from its graphics memory allocator. With such design, the guest view of global graphics memory space is exactly the same as the host view, and the driver programmed addresses, using guest physical address, can be directly used by the hardware. Address space ballooning is different from traditional memory ballooning techniques. Memory ballooning is for memory usage control, concerning the number of ballooned pages, while address space ballooning is to balloon special memory address ranges.

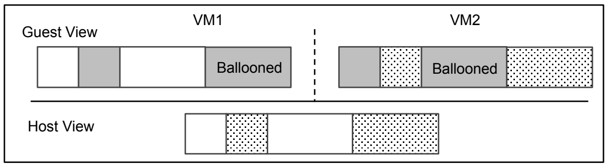


Figure 7: Graphics memory with address space ballooning

Another benefit of address space ballooning is to directly use the guest command buffer, without any address translation overhead, for direct GPU execution. It simplifies the implementation a lot, by eliminating the need of the shadow command buffer, in addition to performance guarantee. However, such scheme may be susceptible to security violation. We address this issue with smart shadowing, by auditing and protecting the command buffer from malicious attack, which is discussed in subsection 3.5.

Per-VM *local graphics memory*: gVirt allows each VM to use the full *local graphics memory* spaces, of its own, similar to the virtual address spaces on CPU. The *local graphics* memory spaces are only visible to the render engine in the GPU. So, any valid *local graphics memory* address programmed by a VM can be used directly by the GPU. The mediator switches the *local graphics memory* spaces, between VMs, when switching the render ownership.

3.4. GPU Page Table Virtualization

gVirt virtualizes the GPU page tables with shared shadow *global page table* and per-VM shadow *local page table*.

Shared shadow *global page table*: To achieve resource partitioning and address space ballooning, gVirt implements shared shadow *global page table* for all VMs. Each VM has its own guest *global page table*, translated from the graphics memory page number to the Guest memory Page Number (GPN). Shadow *global page table* is then translated from the graphics memory page number to the Host memory Page Number (HPN). The shared shadow *global page table* maintains the translations for all VMs, to support concurrent accesses from the CPU and GPU concurrently. Therefore, gVirt implements a single, shared shadow *global page table,* by trapping guest PTE updates, as shown in Figure 8. The *global page table*, in MMIO space, has 512K PTE entries, each pointing to a 4KB system memory page, so in overall creates a 2GB *global graphics memory* space. gVirt audits the guest PTE values, according to the address space ballooning information, before updating the shadow PTE entries.

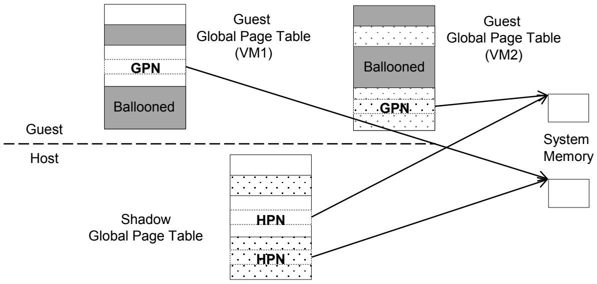


Figure 8: Shared shadow *global page table*

Per-VM Shadow *local page tables*: To support pass-through of *local graphics memory* access, gVirt implements per-VM shadow *local page tables*. The local graphics memory is only accessible from the render engine. The *local page tables are* two-level paging structures, as shown in Figure 9. The first level Page Directory Entries (PDEs), located in the *global page table*, points to the second level Page Table Entries (PTEs), in the system memory. So, guest access to the PDE is trapped and emulated, through the implementation of shared shadow *global page table*. gVirt also write-protects a list of guest PTE pages, for each VM, as the traditional shadow page table approach does [15][25] . The mediator synchronizes the shadow page with the guest page, at the time of write-protection page fault, and switches the shadow *local page tables* at render context switches.

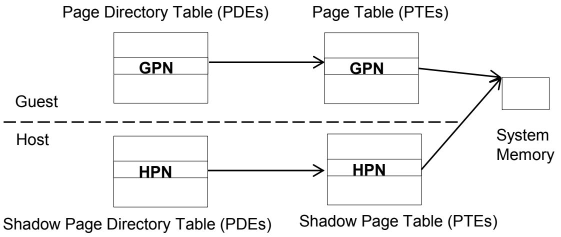


Figure 9: Per-VM shadow *local page table*

3.5. Security

Pass-through is great for performance, but it must meet the following criteria for secure isolation. First, a VM must be prohibited from mapping unauthorized graphics memory pages. Second, all the GPU registers and commands, programmed by a VM, must be validated to only contain authorized graphics memory addresses. Last, gVirt needs to address denial-of-service attacks, for example, a VM may deliberately trigger lots of GPU hangs.

3.5.1. Inter-VM Isolation

Isolation of CPU accesses: CPU accesses to privileged I/O registers and PTEs are trap-and-emulated, under the control of the mediator. Therefore a malicious VM can neither directly change the physical GPU context, nor map unauthorized graphics memory. CPU access to frame buffer and command buffer is also protected, by the EPT.

On the other hand, gVirt reuses the guest command buffer, for the GPU to execute directly for performance, as mentioned in subsection 3.3, but, it may violate isolation, for example, a malicious command may contain an unauthorized graphics memory address. gVirt solves the problem with smart shadowing as detailed in subsection 3.5.2.

Isolation of GPU accesses: gVirt audits graphics memory addresses, in registers and commands, before the addresses are used by the GPU. It is implemented at the time of trap-and-emulating the register access, and at the time of command submission.

Denial-of-service attack: gVirt uses the *device reset* feature, widely supported in modern GPUs, to mitigate the deny-of-service attacks. The GPU is so complex, that an application may cause the GPU to hang for many reasons. So, modern GPUs support *device reset* to dynamically recover the GPU, without the need to reboot the whole system. gVirt uses this capability to recover from a variety of GPU hangs, caused by problematic commands from VMs. In the meantime, upon the detection of a physical GPU hang, gVirt also emulates a GPU hang event, by removing all the VMs from the run queue, allowing each VM to detect and recover accordingly. A threshold is maintained for every VM, and a VM is destroyed if the number of GPU hangs exceeds the threshold.

3.5.2. Command Protection

Balancing performance and security is challenging for full GPU virtualization. To guarantee no unauthorized address reference from the GPU, gVirt audits the guest command buffer at the time of command submission. However there exists a window, between the time when the commands are submitted and when they are actually executed, so a malicious VM may break the isolation by modifying the commands within that window. General shadowing mechanism, such as the shadow page table [15][25], may be applied. However, it is originally designed for the case where the guest content is frequently modified. It may bring large performance overhead and additional complexity in gVirt.

The programming models of the command buffers actually differ from that of the page tables. First, the primary buffer, structured as a ring buffer, is statically allocated with limited page number (32 pages in Linux and 16 pages in Windows), and modification to submitted ring commands (from *head* to *tail*) is not allowed, per the hardware specification. It may be efficient enough to copy only the submitted commands to the shadow buffer. Second, the batch buffer pages are allocated on demand, and chained into the ring buffer. Once the batch buffer page is submitted, it will unlikely be accessed until the page is retired. Shadow buffer can be avoided for such one-time usage.

gVirt implements a smart shadowing mechanism, with different protecting schemes for different buffers, by taking advantage of their specific programming models. That is: ***Write-Protection*** to the batch buffer, which is unlikely modified (so, the write emulation cost is very limited), and ***Lazy-Shadowing*** for the ring buffer, which is small in size and can be copied from the guest buffer to the shadow buffer with trivial cost.

***Lazy-shadowing* to the ring buffer:** gVirt uses a lazy shadowing scheme to close the attack window on the ring buffer. gVirt creates a separate ring buffer, that is, the shadow ring buffer, to convey the actual commands submitted to the GPU. Guest submitted commands are copied from the guest ring buffer to the shadow ring buffer on demand, after the commands are audited. Note that only the commands submitted to the GPU, are shadowed here. Guest access remains passed through to the guest ring buffer, without the hypervisor intervention. The shadow buffer lazily synchronizes with the guest buffer, when the guest submits new commands. The shadow buffer is invisible to a VM, so there is no chance for a malicious VM to attack.

***Write-Protection* to the batch buffer:** The batch buffer pages are write-protected, and the commands are audited before submitting to the GPU for execution, to close the attack window. The write-protection is applied per page on demand, and is removed after the execution of commands in this page is completed by the GPU, which is detected by tracking the advance of ring head. Modification to the submitted commands is a violation of the graphics programming model per specification, so any guest modification to the submitted commands is viewed as an attack leading to the termination of the VM. In the meantime, the command buffer usage may not be page aligned, and the guest may use the free sub-page space for new commands. gVirt tracks the used and unused space of each batch buffer page, and emulates the guest writes to the unused space of the protected page for correctness.

Lazy-shadowing works well for the ring buffer. It incurs an average number of 9K command copies per second, which is a minor cost to a modern multi-GHz CPU. In the meantime, *Write-Protection* works well for the batch buffer, which protects ~1700 pages with only ~560 trap-and-emulations per second, on average.

3.6. Optimization

An additional optimization is introduced to reduce the trap frequency, with minor modifications to the native graphics driver. According to the hardware specification, the graphics driver has to use a special programming pattern at the time of accessing certain MMIO, with up to 7 additional MMIO register accesses [12][13], to prevent the GPU from entering power saving mode. It doesn't incur an obvious cost in the native world, but it may become a big performance challenge, in gVirt, due to the induced mediation overhead. Our GPU power management design gives us a chance to optimize: gVirt relies on Dom0 to manage the physical GPU power, while the guest power management is disabled. Based on this, we optimize the native graphics driver, with a few lines (10 LOC change in Linux) of changes, to skip the additional MMIO register accesses, when it runs in the virtualized environment. This optimization reduces the trap frequency by 60%, on average.

The graphics driver identifies whether it is in a native environment or a virtualization environment, by the information in gVirt_info MMIO window (refer to subsection 3.1). The definition of gVirt_info has been pushed into the GPU hardware specification, so backward compatibility can be followed by future native graphics driver and future GPU generations.

3.7. Discussion

Architecture independency: Although gVirt is currently implemented on Intel Processor Graphics, the principles and architecture can also be applied to different GPUs. The notion of *frame buffer, command buffer*, *I/O registers,* and *page tables*, are all abstracted very well in modern GPUs. Some GPUs may use on-die graphics memory, however, the graphics memory resource partitioning and address space ballooning mechanism, used in gVirt, are also amendable to those GPUs. In addition, the shadowing mechanism, for both the page table and command buffer, is generalized for different GPUs as well. The GPU scheduler is generic, too, while the specific context switch sequence may be different.

Hypervisor portability: It is easy to port gVirt to other hypervisors. The core component of gVirt is hypervisor agnostic. Although the current implementation is on a type-1 hypervisor, we can easily extend gVirt to the type-2 hypervisor, such as KVM [17], with hooks to host MMIO access (Linux graphics driver). For example, one can register callbacks on the I/O access interfaces, in the host graphics driver, so the mediator can intercept and emulate the host driver accesses to the privileged GPU resources.

VM scalability: Although partitioning graphics memory resource may limit scalability, we argue it can be solved in two orthogonal ways. The first way is to make better use of the existing graphics memory, by implementing a dynamic resource ballooning mechanism, with additional driver cooperation, to share the graphics memory among vGPUs. The other way is to increase available graphics memory resource, by adding more graphics memory in future generation GPUs.

Scheduling dependency: An additional challenge, of *full GPU virtualization*, is the dependency of engines, such as 3D, blitter, and media. The graphics driver may use semaphore commands, to synchronize shared data structures among the engines, while the semaphore commands may not be preempted. It then brings the issue of inter-engine dependency, and leads to the gang scheduling policy in gVirt, to always schedule all engines together; however, it impacts the sharing efficiency. We argue this limitation can be addressed, with a hybrid scheme combining both per-engine scheduling and gang scheduling, through constructing an inter-engine dependency graph, when the command buffers are audited. Then, GPU scheduler can choose per-engine scheduling and gang scheduling policies dynamically, according to the dependency graph.

4. Evaluation

We run 3D and 2D workloads in both Linux and Windows VMs. For Linux 3D workloads, gVirt achieves 89%, 95%, 91%, and 60% of native performance in LightsMark, OpenArena, Nexuiz, and UrbanTerror, respectively. For Linux 2D workloads, gVirt achieves 81%, 35%, 28%, and 83% of native performance, in firefox-asteroids, firefox-scrolling, midori-zoomed, and gnome-system-monitor, respectively. For Windows workloads, gVirt achieves 83%, 80%, and 76% of native performance, running 3DMark06, Heaven3D, and PassMark2D, respectively. In the meantime, gVirt scales well without a visible performance drop, up to 7 VMs.

4.1. Configuration

The hardware platform includes the 4th generation Intel Core processor with 4 CPU cores (2.4Ghz), 8GB system memory, and a 256GB Intel 520 series SSD disk. The Intel Processor Graphics, integrated in the CPU socket, supports a 2GB *global graphics memory* space and multiple 2GB *local graphics memory* spaces.

We run 64bit Ubuntu 12.04 with a 3.8 kernel in both Dom0 and Linux guest, and 64-bit Windows 7 in Windows guest, on top of Xen version 4.3. Both Linux and Windows runs native graphics driver with virtualization extension (refer to subsection 3.1). Each VM is allocated with 4 VCPUs and 2GB system memory. The *global graphics memory* resources are evenly partitioned among VMs, including Dom0. For example, the guest is partitioned with 1GB *global graphics memory* in the 1-VM case, and 512MB in the 3-VM case, respectively.

We use the Phoronix Test Suite [28] 3D benchmark including LighsMark, OpenArena, UrbanTerror, and Nexuiz, and Cairo-perf-trace [4] 2D benchmark including firefox-asteroids (firefox-ast), firefox-scrolling (firefox-scr), midori-zoomed (midori), and gnome-system-monitor (gnome), as the Linux benchmarks. In subsection 4.5, we run Windows 3DMark06 [1], Heaven3D [11] and PassMark2D [26] workloads. All benchmarks are run in full screen resolution (1920x1080). We compare gVirt to the native, the direct pass-through (based on Intel VT-d), and also to an API forwarding solution (VMGL [9]). We didn't collect the software emulation approach, since it has already been proved infeasible for modern GPU virtualization [9][22].

Three gVirt configurations are examined to show the merits of individual technologies incrementally.

- *gVirt_base:* baseline gVirt without smart shadowing and trap optimization
- *gVirt_sec*: gVirt_base with smart shadowing
- *gVirt_opt*: gVirt_sec with trap optimization

4.2. Performance

Figure 10 shows the performance of both Linux 3D and 2D workloads normalized to native. 3D workloads are GPU intensive except UrbanTerror. gVirt_base achieves 90%, 94%, 89%, and 47% of native performance for LightsMark, OpenArena, Nexuiz, and UrbanTerror, respectively. UrbanTerror is both CPU and GPU intensive, so it suffers from mediation cost more than the others.

For Linux 2D workloads, gVirt_base achieves 63% and 75% of native performance, for firefox-asteroids (firefox-ast) and gnome-system-monitor (gnome), respectively. However, it reaches only 12% and 15% of native performance, for firefox-scrolling (firefox-scr) and midori-zoomed (Midori) workloads, respectively. This is because they are both CPU and GPU intensive, incurring an up to 61K/s trap frequency, resulting in a very high mediation cost, explained in subsection 4.3.

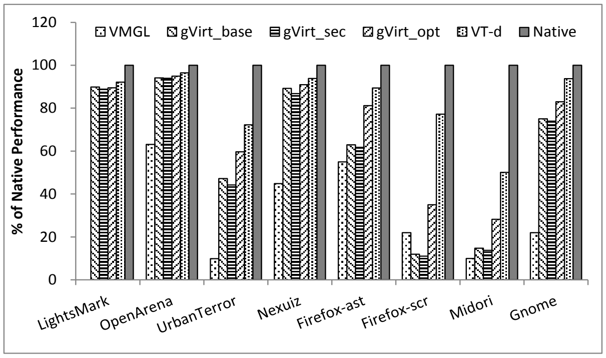


Figure 10: Performance running 3D and 2D workloads

gVirt_sec incurs an average 2.6% and 4.3% performance overhead in 3D and 2D workloads, respectively, much more efficient than a traditional shadowing approach [15][25]. It demonstrates that the smart shadowing scheme can protect the command buffer very effectively, taking advantage of the GPU programming model.

gVirt_opt further improves the performance, up to 214% and 35%, in 2D and 3D workloads, respectively, by optimizing the native graphics driver to reduce the trap frequency. Firefox-scrolling and midori-zoomed achieves the most obvious increase in 2D workloads, by 214% and 104%, respectively. This is because they trigger very high access frequency of I/O registers (54k/s and 40k/s), so they benefit more from trap optimization. In 3D workloads, gVirt with optimization achieves 89%, 95%, 91%, and 60% of native performance, in LightsMark, OpenArena, Nexuiz, and UrbanTerror, respectively. The performance of gVirt is very close to VT-d with direct GPU pass-through. VMGL performs much worse than gVirt, with only 13% of native performance (vs. 60% in gVirt) in UrbanTerror, average 29% of native performance (vs. 57% in gVirt) in 2D workloads, and it fails to run LightsMark.

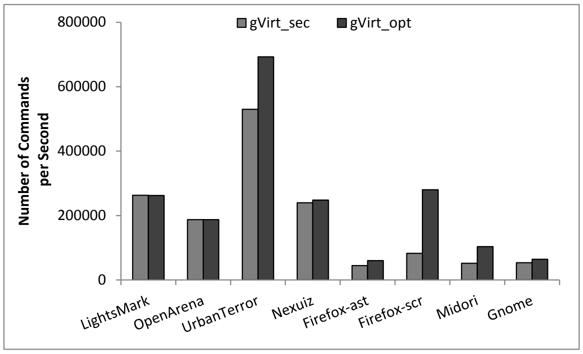


Figure 11: gVirt handles up to 238% more commands, per second, with trap optimization

Furthermore, Figure 11 shows the number of submitted commands per second, with and without trap optimization. UrbanTerror submits 31% more commands per second, with optimization, matching the 35% performance improvement in Figure 10. In firefox-scrolling and midori-zoomed, gVirt handles 238% and 99% more commands per second, with optimization, matching the 214% and 104% performance increase in Figure 10, as well.

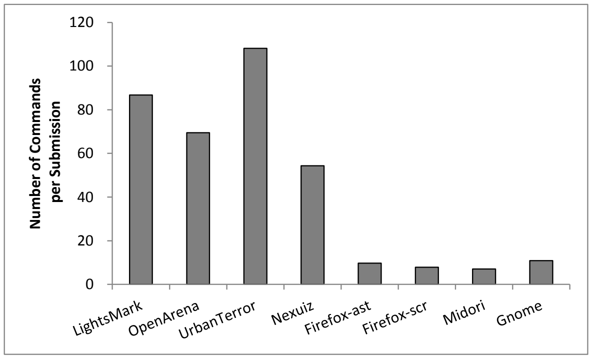


Figure 12: gVirt handles average 8X more commands, per submission, in 3D workloads

We also compare the number of commands per submission, between 3D and 2D workloads, as shown in Figure 12. On average, 3D workloads submit 8X more commands, in every submission, compared to 2D workloads. As a result, 3D workloads induce less mediation overhead per command and achieve better performance.

4.3. Overhead Analysis

We categorize the trap events of gVirt into 4 groups: power management registers (PM) accesses, tail register accesses of ring buffer (Tail), PTE accesses (PTE), and other accesses (Others).

Figure 13 illustrates the break-down of the trap events in gVirt_sec. For 3D workloads, there are around 23K, 22K, 27K, and 33K trap events per second, when running LightsMark, OpenArena, UrbanTerror and Nexuiz, respectively. Among them, 'PM' register access dominates, accounting for up to 67%, 65%, 72%, and 61% of the total trap events, because Linux graphics driver accesses additional PM registers (up to 7) to protect the hardware from entering power saving mode, per hardware specification, when accessing certain registers [12][13]. Tail register access counts for 13%, 12%, 17%, and 13% of the total trap events, respectively. Similarly, 'PM' register access in 2D workloads dominates the trap events as well, accounting for 76% of the total trap rate, on average. 2D workloads has an average 37K/s trap events, 42% higher than that in the 3D workload (26K/s).

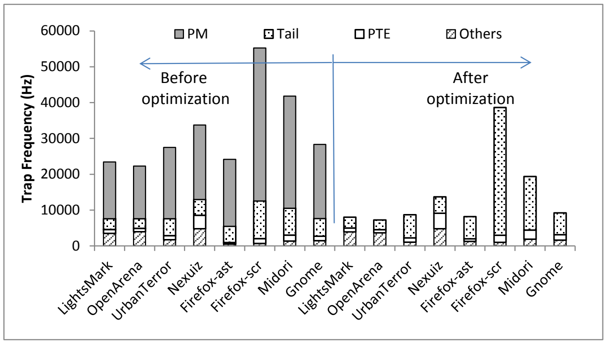


Figure 13: Break-down of the trap frequency, before and after optimization

gVirt_opt reduces the trap events dramatically, as shown in Figure 13. The trap event reduction comes from the removal of all the PM register accesses, which is unnecessary to vGPUs (The real power is managed by Dom0). After the optimization, gVirt reduces the trap rate by an average 65% and 54% for 3D and 2D workloads, respectively. Firefox-scrolling and midori-zoomed have more tail updates, from 19% and 18%, respectively, to 92% and 76% of total traps, which matches the much improved performance (214% and 104% higher), as seen in Figure 10.

The overhead of the smart shadowing scheme is very limited. gVirt_sec copies average 5K and 12.8K ring buffer commands (typically 1-5 double-words per command), per second, for 3D and 2D workloads, respectively. It write-protects an average of 2000 and 1300 batch buffer pages, along with ~870 and ~150 write emulations due to unaligned batch buffer usages, per second, in 3D and 2D workload, respectively. The CPU cycles spent for smart shadowing are trivial for a modern multi-GHz processor. gVirt_sec incurs very limited virtualization overhead, matching the performance shown in subsection 4.2.

4.4. Scalability

Figure 14 presents the scalability of gVirt (gVirt_opt), with all features and optimizations, from 1 VM to 7 VMs, running the same workloads in all VMs, with performance normalized to 1 VM case. For LightsMark, OpenArena and Nexuiz, the performance remains almost flat, demonstrating that the GPU computing power can be efficiently shared among multiple VMs. In UrbanTerror, we see an 8% performance increase, from 1vm to 7vm, because CPU parallelism helps UrbanTerror, which is both GPU and CPU intensive. For 2D workloads, firefox-asteroids and gnome-system-monitor doubles performance from 1vm to 3vm, because they are more CPU intensive (relatively low access rate to GPU resources), so adding more VMs improves performance. The physical CPU cores saturate eventually, so the performance remains flat, from 3vm to 7vm. In all cases, the performance of gVirt doesn't drop obviously with more VMs, demonstrating very good scalability.

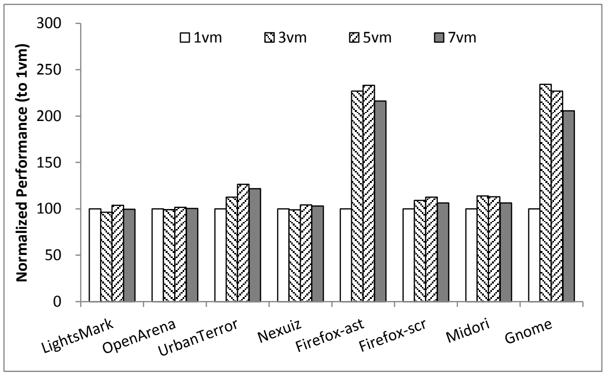


Figure 14: Scalability of gVirt

4.5. Windows

Figure 15 shows the performance of Windows graphics workloads, with smart shadowing and trap optimization (gVirt_opt). We didn't run the baseline gVirt performance, because the Windows driver we received from the production group has already implemented the virtualization extension, without an option to turn off the trap optimization. For 3DMark06 and Heaven3D, gVirt achieves 83% and 81% of native performance, respectively, which are very close to the VT-d performance (85% and 87% of native performance). In PassMark2D, gVirt achieves 76% of native performance, better than that of the Linux 2D workloads (average 57% of native performance), because Windows 2D workload incurs only an average 6k traps per second, 57% less than that of Linux 2D workloads, and therefore less mediation cost. VMGL doesn't support Windows guest.

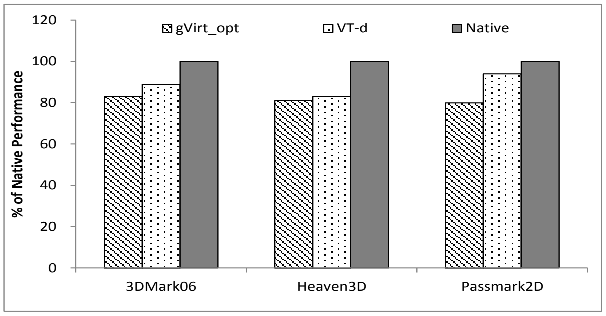


Figure 15: Performance running Windows 3D/2D workloads

Further experiments show that smart shadowing brings only 1.1% and 4.8% performance overhead for Windows 3D and 2D workloads, respectively. It write-protects an average 3600 batch buffer pages, and copies about 10k ring buffer commands, per second, demonstrating that the smart shadowing scheme can protect the command buffer very efficiently, taking advantage of the GPU programming model, in Windows as well.

5. Related Work

Emulating a full-fledged GPU, purely through software, is impractical due to complexity and extremely low performance. Qemu [7] emulates only the legacy VGA cards, with a para-virtualized frame buffer [20] to accelerate 2D specific frame buffer accesses.

API forwarding is the most widely studied technique for GPU virtualization, so far. VMGL [9], Xen3D [3] and Blink [14] install a new OpenGL library in Linux VM, forwarding OpenGL API calls to the host graphics stack for acceleration.. GViM [31], vCUDA [18] and LoGV [23] implement similar API forwarding techniques, focusing on GPGPU computing. VMware's Virtual GPU [22] emulates a virtual SVGA device, implementing a private SVGA3D protocol to forward the DirectX API calls. However, API forwarding faces the challenge of supporting full features, due to the complexity of intrusive modification in the guest graphics stack, and incompatibility between the guest and host graphics stack.

Device Pass-through achieves high performance in I/O virtualization. VT-d [5][37] translates memory addresses of DMA requests, allowing the GPU to be assigned to a single VM. SR-IOV [27] extends the VT-d technology with a device hardware extension. It has been widely used in the network device [36], by creating multiple virtual functions, which can be individually assigned to VMs. VPIO [19] introduces a "virtual pass-through I/O" concept, where the guest can access the hardware resource directly, mostly of the time, for legacy network cards (NE2000 and RTL8139). They either sacrifice the sharing capability, or are not yet available to modern GPUs.

GPU scheduler is well explored. Kato [29] *et al.* implements a priority-based scheduling policy for multi-tasking environment, based on monitoring GPU commands issued from user space. Kato [30] *et al.* further extends that policy with a context-queuing scheme and virtual GPU support. Gupta [32] *et al.* proposes CPU and GPU coordinated scheduling, with a uniform resource usage model to describe the heterogeneous computing cores. Ravi [33] *et al.* implements a scheduling policy, based on affinity score between kernels, when consolidating kernels among multiple VMs. Becchi [21] *et al.* proposes a virtual memory based runtime, supporting flexible scheduling policies, to dynamically bind applications to a cluster of GPUs. Menychtas [16] *et al.* proposes a disengaged scheduling policy, having the kernel grant application access to the GPU, based on infrequent monitoring of the application's GPU cycle use. They were not applied to *full GPU virtualization*, yet.

NVIDIA GRID [24] allows each VM's GPU commands to be passed directly to the GPU for acceleration. A vGPU manager shares the GPU based on time slices. It looks similar to gVirt in some ways; however there is no public information on technical details, or open access to the project.

6. Conclusion and Future Work

gVirt is a *full GPU virtualization* solution with mediated pass-through, running a native graphics driver in the VM, with a good balance among performance, features, and secure sharing capability. We introduce the overall architecture, with the policies of mediation and pass-through base on the access patterns to the GPU interfaces. To ensure efficient and secure graphics memory virtualization, we propose graphics memory resource partitioning, address space ballooning, shared shadow global page table, per-VM shadow local page table, smart shadowing mechanism, and additional optimization to remove the unnecessary trap events. gVirt presents a vGPU instance to each VM, with full features, based on trap-and-emulating privileged operations. Such full GPU virtualization solution allows the native graphics driver to be run inside a VM. We also reveal that different programming model of applications might introduce different trap frequency and therefore different virtualization overhead. Lastly, gVirt is an open source implementation, so it provides a solid base for follow-up GPU virtualization research.

As for future work, we will focus on the areas of portability, scalability, and scheduling areas, as discussed in subsection 3.7, in addition to fined-grained QoS scheduling policy. In the meantime, we will evaluate hardware assistance to further reduce the mediation cost. Hypervisor interposition features are also interesting to us, for example, supporting VM suspend/resume and live migration [6]. With gVirt as the vehicle, we will extend *full GPU virtualization* to more usages, in desktop, server, and mobile devices, to exploit specific challenges in different use cases.

References

[1] 3DMark06. http://www.futuremark.com

[2] Amazon GPU instances. http://aws.amazon.com/ec2/instance-types/

[3] C. Smowton. Secure 3D graphics for virtual machines. In EuroSEC'09: Proceedings of the Second European Workshop on System Security. ACM, 2009, pp. 36-43.

[4] Cairo-perf-trace. http://www.cairographics.org

[5] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. Intel Technology Journal, 10, August, 2006.

[6] E. Zhai, G. D. Cummings, and Y. Dong. Live migration with pass-through device for linux vm. In *Proc.* OLS (2008)

[7] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc.* USENIX ATC (2005)

[8] gVirt. https://github.com/01org/XenGT-Preview-kernel, https://github.com/01org/XenGT-Preview-xen, https://github.com/01org/XenGT-Preview-qemu

[9] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. D. Lara. VMM-independent graphics acceleration. In *Proc.* VEE (2007), pp. 33-43

[10] H. Shan, K. Tian, Y. Dong, and D. Cowperthwaite. XenGT: a Software Based Intel Graphics Virtualization Solution. Xen Project Developer Summit (2013)

[11] Heaven3D. http://unigine.com/products/heaven

[12] Intel Graphics Driver. http://www.x.org/wiki/IntelGraphicsDriver/

[13] Intel Processor Graphics PRM. https://01.org/linuxgraphics/documentation/2013-intel-core-processor-family

[14] J. G. Hansen. Blink: Advanced display multiplexing for virtualized applications. In *Proc*. NOSSDAV (2007)

[15] K. Adams and O.Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. In *Proc.* ASPLOS (2006)

[16] K. Menychtas, K. Shen, and M. L. Scott. Disengaged Scheduling for Fair, Protected Access to Fast Computational Accelerators. In *Proc.* ASPLOS (2014)

[17] KVM. www.linux-kvm.org/

[18] L. Shi, H. Chen, and J. Sun. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines. In Proc. IEEE IPDPS (2009)

[19] L. Xia, J. Lange, P. Dinda, and C. Bae. Investigating virtual passthrough I/O on commodity devices. In *Proc.* ACM SIGOPS (2009), pp. 83-94

[20] M. Armbruster. The Xen Para-virtualized Frame Buffer. Xen Summit (2007).

[21] M. Becchi, K. Sajjapongse, I. Graves, A. Procter, V. Ravi, and S. Chakradhar. A virtual memory based runtime to support multi-tenancy in clusters with GPUs. In *Proc.* HPDC (2012), pp. 97-108

[22] M. Dowty and J. Sugerman. GPU virtualization on VMware's hosted I/O architecture. In *Proc.* ACM SIGOPS (2009), pp. 73-82

[23] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. LoGV: Low-overhead GPGPU virtualization. In *Proc.* IEEE Workshop on Frontiers of Heterogeneous Computing (2013).

[24] NVIDIA GRID. http://on-demand.gputechconf.com/gtc/2013/presentations/S3501-NVIDIA-GRID-Virtualization.pdf

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc.* ACM SOSP (2003), pp. 164-177

[26] PassMark2D. http://www.passmark.com

[27] PCI SIG. I/O virtualization. http://www.pcisig.com/specifications/iov

[28] Phoronix Test Suites. http://phoronix-test-suite.com

[29] S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa. TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments. In *Proc.* USENIX ATC (2011)

[30] S. Kato, M. McThrow, C. Maltzahn, and S. BrandtGdev. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc.* USENIX ATC (2012)

[31] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan. GViM: GPU-accelerated virtual machines. In *Proc.* ACM HPCVirt (2009), pp. 17-24

[32] V. Gupta, K. Schwan, N. Tolia, V.Talwar, and P. Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based system. In *Proc.* USENIX ATC (2011)

[33] V. T. Ravi, M. Becchi, G. Agrawal, and S. Chakradhar. Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework, In *Proc.* HPDC (2011), pp. 217-288

[34] VMware Horizon View. http://www.vmware.com/products/horizon-view/

[35] XenClient. http://www.citrix.com/products/xenclient/overview.html

[36] Y. Dong, X. Yang, X. Li, J. Li, K. Tian, and H, Guan. High performance network virtualization with SR-IOV. In *Proc.* IEEE HPCA (2010), pp. 1-10

[37] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, Y. Jiang, Towards high-quality I/O virtualization. SYSTOR 2009

vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment

Fei Meng[†], *Li Zhou*[¶,*] *Xiaosong Ma*[†§], *Sandeep Uttamchandani*[‡] *and Deng Liu*[◇*]

[†]North Carolina State University, fmeng@ncsu.edu
[¶]Facebook Inc., lzhou@fb.com [§]Qatar Computing Research Institute, xma@qf.org.qa
[‡]VMware Inc., suttamchandani@vmware.com [◇]Twitter Inc., dengl@twitter.com

Abstract

Server Flash Cache (SFC) is increasingly adopted in virtualization environments for IO acceleration. Deciding the optimal SFC allocation among VMs or VM disks is a major pain-point, dominantly handled manually by administrators. In this paper, we present *vCacheShare*, a dynamic, workload-aware, policy-driven framework for continuous and automated optimization of SFC space partitioning. Its decision-making is based on multiple IO access characteristics. In particular, vCacheShare adopts a cache utility model that captures both longer-term locality behavior and transient locality spikes.

This paper validates the growing applicability of analytical programming techniques to solve real-time resource management problems, traditionally addressed using heuristics. We designed vCacheShare to coordinate with typical VM mobility events and implemented it within the widely used ESXi hypervisor. We performed extensive evaluation using 13 representative enterprise IO workloads, one IO benchmark, and two end-to-end deployment test cases targeting Virtual Desktop Infrastructure (VDI) and data warehousing scenarios respectively. Our results verified the advantage of vCacheShare over implicit management schemes such as global LRU, and confirmed its self-adaptive capability.

1 Introduction

Solid State Disks (SSDs) are being increasingly used in virtualized environments as an SFC (Server Flash Cache) to accelerate I/O operations of guest Virtual Machines (VMs). However, growing CPU bandwidth and memory capacities are enabling higher VM-to-server consolidation ratios, making SFC management a nightmare. The onus of proportional allocation of SFC space among VMs is handled manually by administrators today, based on heuristics and oftentimes simply guesswork. Besides, allocations should not be one-time activities, but continuously optimized for changing characteristics of workloads, device service times, and configuration events related to VM mobility. In order to effectively leverage flash for its $/IOPS advantages and to address existing performance bottlenecks within the infrastructure, it is critical to extend the hypervisor to automate and continuously optimize the space management of SFC, similar to other hardware resources.

The primary use-case of SFC is to speed up applications running inside VMs. Scenarios with performance bottleneck of the backend storage arrays benefit the most, delaying capital investments in provisioning of new hardware. At the hypervisor level, SFC space optimization translates to reducing the I/O latency of VMs, which are prioritized based on administrator input. The optimization needs to take into account multiple dimensions:

- VM priority: VMs have different importance depending on applications running inside.
- Locality of reference: A VM running low-locality workload(s) does not benefit as much from caching and should therefore receive smaller allocation.
- I/O access characteristics: A VM running write-heavy workload(s) may receive smaller benefit and incur higher cost with flash caching, due to SSD devices' asymmetric read-write performance and write durability concerns.
- Backend storage device service times: A VM Disk on faster (or less busy) backend storage benefits less from SFC allocation.
- Configuration events: Hypervisors are optimized to continuously monitor and optimize the placement of resources from a virtualized pool of hardware. Guest VMs can be migrated to other servers without application down time, with ready functionality available such as VMware vMotion.

Current commercial solutions [1, 2, 3, 4, 5, 6] require administrators to carve static space allocations at the time of enabling SFC for VMs. Meanwhile, memory virtualization techniques within hypervisors are not directly applicable to SFC space management. The reasons are two-fold. First, memory resources are explicitly requested by users and the hypervisor has to satisfy

[*] With VMware during this work

such requests with reasonable latency, while cache resources are transparent to applications and allocated by the hypervisor for performance enhancement. Second, typical VMs expect their data to be memory-resident in most cases, and heavy swap activities are seen as an indication of intensive memory contention and may subsequently trigger VM migration. In contrast, flash cache is designed as an intermediate level of storage, in order to accelerate IO rather than to accommodate all VMs' combined working set. At the same time, existing CPU cache partitioning techniques [7, 8, 9] will not be able to exploit options available to managing flash-based SSD cache spaces. The latter problem is significantly different in aspects such as response time, space availability for optimization, access patterns, and performance constraints. In general, affordable in-memory book keeping and the much slower storage access enable a much larger solution space and more sophisticated algorithms. Unique flash storage issues such as write endurance also bring additional complexities.

Another alternative is to retrofit IO QoS management techniques [10, 11] for SFC management. IO QoS aims at providing fairness and prioritization among requests serviced under bandwidth contention. This problem is different from SFC management, as the definition of resource contention is much less straight-forward for caching — the criteria here is not "actively used cache space" by the VM, but rather the performance gain through caching based on its IO access locality behavior.

In this paper, we present *vCacheShare*, a dynamic, workload-aware, policy-driven framework for automated allocation of flash cache space on a per-VM or per-VM-Disk basis. vCacheShare combines the dimensions of data locality, IO operation characteristics, and device service time into a self-evolving *cache utility model*. It approaches flash cache allocation/partitioning as a constrained optimization problem with the objective of maximizing the cumulative cache utility, weighted by administrator-specified VM priorities.

Contributions This paper addresses the pain-point of optimal space partitioning of SFC in a virtualization environment. Our goal is to develop a mathematical optimization solution for SFC management. With the trend of CPU threads per socket doubling every 12-18 months [12], we think the time has come for resource management techniques to adopt such strategies for dynamic multi-dimensional optimization.

We consider the major contributions of this work as follows: **(1)** We addressed a unique challenge in managing time-varying IO behavior common to many VM workloads, namely, the conflict in long-term behavior analysis required for accurate cache hit ratio estimation and fast response required for handling transient locality bursts. In response, we propose a novel cache utility modeling approach that takes into consideration both long-term reuse patterns and short-term reuse intensity levels. **(2)** We designed vCacheShare, a cost-effective, adaptive framework that scales in today's high-VM-density environments, based on the proposed techniques. **(3)** We implemented our prototype within a widely used commercial hypervisor, VMware ESXi 5.0, and performed extensive evaluation using a combination of 10+ real-world traces, one IO benchmark, and two workload deployment tests with up to 100 VM instances.

2 A Bird's Eye-view of vCacheShare

2.1 Target Environment

Typically in a datacenter using virtualization, multiple hosts share the same SAN/NAS storage to utilize the rich set of functionality provided by the hypervisor. The hosts sharing storage form a *cluster* within the datacenter. In such an environment, a file or block storage is visible to the *Virtual Machine (VM)* as a *Virtual Machine Disk (VMDK)*. One VM could have multiple VMDKs just as a physical machine may have several physical disks. The hypervisor (sometimes referred to as *server*) internally tracks VMs and VMDKs using Universally Unique Identifiers (UUIDs). The UUIDs for VMs and VMDKs are unique cluster-wide. In our target environment, both VMs and VMDKs can be moved non-disruptively across physical resources — commonly referred to as vMotion and Storage vMotion respectively.

vCacheShare adopts the *write around* cache policy, i.e., writes bypass the SFC and directly go to the backend disks. There are several reasons to focus on read intensive workloads. First, there are multiple layers of memory cache above SFC, which are shown to have reduced the read-after-write percentage [13] and consequently weakened the benefit of write caching. Second, write around prolongs the SFC lifetime by reducing writes and prioritizing data to be re-read. Finally, it simplifies design and implementation by relaxing consistency requirement. In fact, several commercial SFC solutions [2, 4] started from write around or write through, including VMware's recent vFRC (Flash Read Cache).

2.2 vCacheShare Overview

vCacheShare decides the SFC space allocation sizes on per-VM or per-VMDK basis, based on runtime analysis of the VMs/VMDKs' IO access characteristics, along with hardware/administrative settings.

Figure 1 depicts vCacheShare's location in our target environment, as well as its internal architecture. The vCacheShare decision making workflow requires the coordination among its major components: (1) *monitor*, which intercepts IO requests and logs information about them, (2) *analyzer*, which periodically processes trace

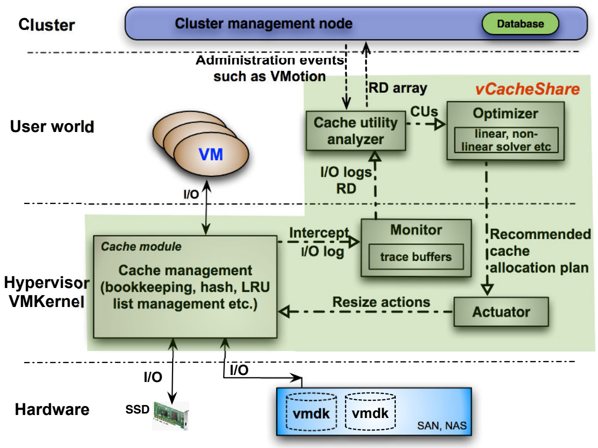


Figure 1: vCacheShare architecture

data logged by the monitor and extracts per-VMDK access characteristics, (3) *optimizer*, which, based on the analyzer-extracted workload characteristics and a *cache utility model*, dynamically optimizes the cache partitioning plan, and (4) *actuator*, which executes the optimized plan. The analyzer and optimizer are implemented as user-level agents, while monitor and actuator are implemented within the VMkernel.

In addition to runtime workload monitoring input, vCacheShare also takes from administrators two key configuration parameters: (1) a per-VMDK IO priority level and (2) per-VMDK cache allocation lower and upper bounds. The lower bound is also referred to as *reserved cache space*. Not all VMs/VMDKs need flash cache enabled: for example, some VMDKs may be provisioned out of an all-flash array.

In the rest of the paper, we base our discussion on per-VMDK cache space allocation, though the techniques and observations also apply to per-VM allocation.

2.3 Event-Driven Analysis and Re-Optimization

The vCacheShare design is well integrated with the hypervisor's VM management. In particular, its workflow has to be coordinated with common VM state transitions and migration operations (for both VMs and storage devices). Table 1 lists vCacheShare's actions triggered by such common VM management activities. Though the events are hypervisor specific, we believe that the vCacheShare workflow and event-handling methodology can be applied to other hypervisors as well.

3 Monitoring

The vCacheShare monitoring module sits within the hypervisor kernel, on the IO path between the VMs and the SFC. It intercepts, records, and analyzes all IO accesses from the VMs on each physical host. For each VMDK, it performs periodic online trace processing to extract the reuse pattern needed by the vCacheShare cache utility model (Section 4).

vCacheShare's trace data collection is performed on a per-VMDK basis. For each cache-enabled VMDK, vCacheShare sets up a circular log buffer in kernel memory for storing trace entries. Their content can be committed to SSD storage regularly and asynchronously to reduce memory usage. The in-memory and on-SSD circular buffer sizes, as per-VMDK configurable parameters, limit the maximum memory/SSD space consumed by vCacheShare IO trace collection. In our experiments, each VMDK has a 4MB on-SSD trace buffer, plus two 64KB in-memory ones for double buffering.

Each trace entry is 15 bytes long, containing 8 fields. Among them, the `VM_UUID` and the `VMDK_UUID` fields identifies the IO request issuer (VM) and the destination device (VMDK), respectively. The `timestamp` field logs the request time. `isRead` records the IO operation type, while `LBA` and `len` define the initial logical block address and IO request size (in terms of blocks). `latency` records the total service time to complete the IO. For each entry, most of the data collection happens before the IO request hits the flash caching module, except for `latency`, which is logged upon IO completion. Finally, `isCached` tags IOs serviced from the flash cache rather than from the backend storage. As measured with our implementation, such trace collection activities add only 4-5 nanoseconds to the overall microsecond- or millisecond-level IO service time.

Note that the above vCacheShare monitoring and trace collection is in addition to the hypervisor's built-in IO profiling. Such profiling gather high-level statistics through approaches such as moving window averages, regarding IOPS, latency, read-write ratio, request sizes, etc., on a per VM or VMDK basis. The IO statistics are averaged over a user configurable monitoring window (default at 20 seconds). Some of these data items, e.g., read percentage and average latency, are used in vCacheShare's cache utility model calculation.

4 Dynamic Cache Utility Analysis

vCacheShare performs its cache space allocation optimization based on *cache utility (CU)* [7] on a per-VMDK basis. CU reflects the effectiveness of allocating SFC space, i.e., the relative performance gain brought by additional SFC space allocation.

Cache utility depends on several factors. Intuitively, a VMDK generates better cache utility if it displays good reference locality or is built on backend storage devices with higher read latency (either due to slower device speed or saturation). In addition, vCacheShare favors read-heavy workloads, both due to the current write around caching design and well-known performance plus endurance problems with flash writes. Such limitations

VM Events	Actions by vCacheShare Framework
VM Power-Off, Suspend, vMotion Source	Trigger optimization to re-allocate; free IO trace buffers for all associated VMDKs
VM Bootstrap, Power-On, Resume	Make initial allocation based on reserved cache space and priority settings; start trace collection
vMotion Destination	Trigger optimization to re-allocate based on IO characteristics migrated with the VMDKs involved in vMotion
Storage vMotion (runtime backend device change)	Suspend analysis/optimization till completion; evict device service latency history; trigger re-optimization upon vMotion completion
VM Fast Suspend, vMotion Stun	Reserve cached data; lock cache space allocation to involved VMs by subtracting allocated size from total available cache size

Table 1: Actions by vCacheShare upon VM Events

make read-intensive access patterns more flash-cache-friendly even with write caching turned on in the future. Finally, vCacheShare takes into account an attribute much less often factored in for cache partitioning: the intensity of re-accesses. This metric considers both locality and access speed, allowing our model to favor workloads that generate more reuse of cached data in unit time.

Device access speed and read-write ratio can be measured rather easily. Hence our discussion focuses on estimating access locality and reuse intensity.

4.1 Measuring Locality

Many prior studies have examined cache hit ratio prediction/estimation based on workload access characteristics [14, 15]. vCacheShare adopts a low-overhead online estimation approach that examines access reuse distance, similar to that used in CPU cache partitioning [14]. However, vCacheShare explores reuse distance for storage cache partitioning, where more sophisticated follow-up algorithms could be explored due to both more resource availability and slower data access rates.

For each identified re-access, where the same block is revisited, the *reuse distance* is measured as the number of distinct blocks accessed between the two consecutive uses of that block [14]. vCacheShare constantly monitors the *distribution of reuse distances*, to measure the temporal locality in accesses to each VMDK. Considering that our current read-only cache design, only read accesses are counted in vCacheShare's reuse distance calculation. Such per-VMDK overall temporal locality for reads is expressed in a CDF (cumulative distance function) of reuse distances.

To dynamically build and update the reuse distance distribution, vCacheShare first extracts relevant data from its IO traces using a two-phase $O(n)$ process, where n is the number of trace entries. This Map-Reduce like process can easily be parallelized to use multiple cores available, if necessary.

vCacheShare maintains one hash table per VMDK. In the "Map" phase, IO traces are scanned into the appropriate hash table. We pre-process IO logs here to address large block IOs and un-aligned 4K accesses. A log entry request for greater than 4KB is divided into multiple 4KB entries, with LBA adjusted for each. This is inline with our target flash cache implementation, where cache book-keeping is done at the 4KB granularity. Based on our analysis of real-world workloads from the SNIA MSR traces [16], only 5-10% of accesses are unaligned, which are discarded in our processing. For each read log entry, a pointer to the entry is inserted into the per-VMDK hash table, using its LBA as the key.

In the "Reduce" phase, vCacheShare scans the values of the per-VMDK hash tables, generating for each a *Reuse Distance Array* (*RD Array*). This is an array of lists that for each LBA, stores its reuse distances between each pair of adjacent accesses. The reuse distance CDF can then be easily constructed using histograms.

As mentioned in Section 3, vCacheShare maintains per-VMDK cyclic trace buffers. The per-VMDK hash tables are updated according to a time stamp marking the oldest valid log entry in the trace buffer. Older entries are discarded from the hash tables, during either insertion or periodic garbage collection sessions. This way, vCacheShare automatically assesses access patterns in a sliding-window manner, with a *sampling window size* equal to the size of the cyclic trace buffer.

4.2 Locality-based Hit Ratio Estimation

With the reuse distance CDF calculated as described above, we can produce a rough cache hit ratio estimation for each VMDK at a given cache size. Assuming an LRU or LRU-like cache replacement algorithm, the hit ratio can be computed by dividing the total number of cache hits (re-accesses whose reuse distance falls under the cache size) by the total number of accesses.

The effectiveness of such estimation, however, depends heavily on the aforementioned sampling window size. Intuitively, if the sampling window is too small, vCacheShare will not be able to fully capture a workload's temporal locality. On the other hand, large sampling windows will produce slower response to workload behavior changes, as new access patterns will be "dampened" by a large amount of older entries. In addition, larger sampling windows require more space and time overhead for trace collection and analysis.

Figure 2 demonstrates the impact of sampling win-

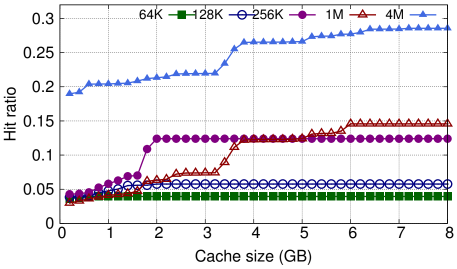


Figure 2: Impact of sampling window size on hit ratio estimation

dow size on hit ratio estimation, through a representative workload, "web", member of the SNIA MSR traces [16]. It shows that in general, larger sampling window sizes produce higher cache hit ratio estimates, and reach hit ratio saturation points later (with exceptions likely due to well known caching anomalies). On the other hand, a reasonably large sampling window size (e.g., 1MB) seems to produce very similar trend in hit ratio growth to a window size a few times larger (4MB). Other read-intensive traces in the MSR collection show similar results. We choose to leave the sampling window size as a tunable parameter. Note that a larger flash device offers more cache space and can benefit from larger sampling windows. This goes well with our design of having SSD-resident cyclic trace buffers, as the same device can accommodate higher space overhead as well.

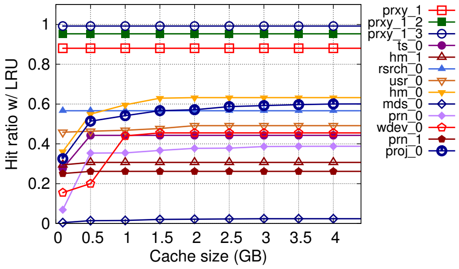


Figure 3: Hit ratio as a function of cache size

To further verify the need of cache partitioning, Figure 3 shows the cache hit ratios of a selected set of representative MSR traces under different cache sizes. Each trace is replayed for the first one million IO accesses. We can clearly see that (1) different workloads from the same enterprise environment have diverse locality properties, and (2) among workloads that possess certain locality, there are large variances in their working set sizes.

4.3 Measuring Reuse Intensity

Trace based temporal locality analysis has been shown to provide quite accurate hit ratio estimation [14, 15]. However, for cache partitioning between concurrent accesses to multiple VMDKs, such analysis fails to consider the *relative speed* of cache accesses. In particular, it does not capture bursts in data reuse. vCacheShare needs to identify and absorb *locality spikes*, caused by common activities such as boot storms in VDI environments [17]. Fast response is especially challenging with a larger sampling window, due to its dampening effect.

For this, we introduce another factor in modeling CU: *reuse intensity*, which measures the **burstiness of cache hits**. This metric captures the fast changing aspect of CU, to bias cache allocation toward VMDKs undergoing locality spikes. More accurately, for $VMDK_i$, its reuse intensity RI_i is defined as

$$RI_i = \frac{S_{total}}{t_w \times S_{unique}} \qquad (1)$$

Here, for a given monitoring time window size t_w, S_{total} and S_{unique} describe the total read access volume and the total size of unique blocks read (i.e., access footprint). E.g., with t_w at 5 minutes, within which 1G blocks are read from $VMDK_i$ accessing 1000 unique blocks, the resulting RI_i will be $\frac{1G}{1000 \times 300s}$. This metric effectively captures the locality-adjusted per-VMDK read throughput: an influx of accesses to new blocks brings similar growth to total access volume and footprint, therefore lowering RI; an influx of re-accesses, on the other hand, increases the former but not the latter, inflating RI.

t_w is another tunable vCacheShare parameter, preferably with relatively small values for better responsiveness. It asserts little computation overhead, and can be maintained as a per-VMDK moving average. In situations where detailed trace collection/analysis is impractical or overly expensive, RI can contribute independently to the vCacheShare CU model as a cheap and less accurate alternative, since it does reflect temporal locality. However, t_w should ideally be decided adaptively based on the rate of locality change measured in real time. We are currently performing follow-up study on devising such a dynamic algorithm. In this paper, we empirically observed that t_w values between 2 and 60 seconds are suitable for the workloads tested. Our experiments used 60 seconds unless otherwise noted.

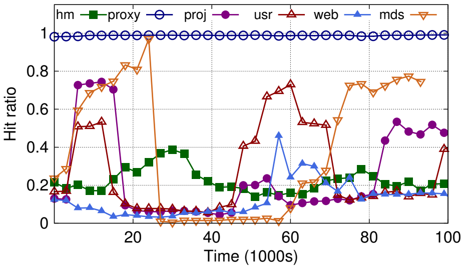


Figure 4: Temporal pattern in cache hit rate; the hit rate is calculated as a moving average

To verify the existence of locality spikes, we examine the temporal locality shifts in the MSR workloads.

Figure 4 plots the hit rate changes for six sample traces along the execution timeline. The hit ratio in y axis is calculated with a 60-second moving window. The results illustrate the time-varying nature of reference locality, as well as the existence of transient locality spikes in multiple traces (such as *mds*, *usr*, and $proj$).

4.4 Summary: Cache Utility Model

Putting everything together, vCacheShare's CU Model for $VMDK_i$ is generated as a function of its estimated cache hit rate $H_i(c)$ (where c is the target cache partition size), reuse intensity RI_i, average target device latency l_i, and read ratio RR_i (fraction of total accesses that are reads): If we expand the variables to show the input of the model, we end up with the following equation:

$$CU_i = l_i \times RR_i \times (H_i(c) + \alpha \hat{RI}_i) \qquad (2)$$

Here $H_i(c)$ generates the estimated hit ratio for $VMDK_i$ at cache partition size c, based on its locality observed in the previous sampling window. $\hat{RI}_i$ gives the reuse intensity observed for the same VMDK, *normalized* to the highest *RI* across all VMDKs in the previous intensity monitoring window. Therefore, both $H_i(c)$ and RI_i have values between 0 and 1. α is included as an additional tuning knob to adjust the relative weight between longer-term, more accurate temporal locality, and short-term locality spikes. Though set to 1 in our evaluation, system administrators can change α to favor persistently cache-friendly workloads or fast-changing, bursty accesses.

5 Optimization

vCacheShare approaches SFC space allocation as a constrained optimization problem. Its optimizer explores different permutations of per-VMDK space allocation values, calculates cache utility as the objective function, and returns the permutation with the global maximum in the search space. More specifically, vCacheShare adopts the following objective function:

$$\sum_{i=1}^{n} priority_i \times CU_i,$$

where n is the number of VMDKs, $priorite_i$ is the user- or administrator-defined priority level (e.g., based on QoS requirement) of $VMDK_i$, and CU_i is the cache utility of $VMDK_i$ defined earlier.

The vCacheShare optimizer will search for the global optimum in the form of a recommended cache allocation plan: $< c_1, c_2, \ldots, c_n >$, which satisfies the following constraints:

$$c_1 + c_2 + \ldots + c_n = C$$
$$c_{min} \leq c_i \leq c_{max}$$

where C is the total available server-side flash cache size, and c_{min}/c_{max} is the administrator-specified per-VMDK cache space lower/upper bound.

This optimization problem is NP-Hard, with much existing research on heuristics such as simulated annealing [18] and hill-climbing [19]. These techniques approximate the optimal solutions via linear, non-linear, or piecewise linear algorithms, among others. We consider such constrained optimization a mature field and beyond the scope of this work. Our prototype implementation uses an open-source simulated annealing tool [18], while vCacheShare is designed to be able to utilize alternative optimization algorithm plug-ins.

6 Execution

The execution module actuates changes in per-VMDK SFC allocation. It also controls the bootstrapping process when vCacheShare is enabled for the first time on a server or when new VMs are added.

For bootstrapping, vCacheShare allocates the administrator-specific per-VMDK cache space lower bound (also referred as *reserved* size) for each enabled VMDK. The rest of the available cache size is then divided among the VMDKs, proportional to their priorities. When a VMDK is first added to a running vCacheShare instance, as its CU estimate is not yet available, it again receives the reserved allocation, by reclaiming space proportionally from existing allocations according to VMDK priorities.

Once bootstrapped, vCacheShare manages SFC as per-VMDK linked lists of blocks, each with a $current\_size$ and a $target\_size$. Upon invocation, the actuator only changes the $target\_size$ for each list, based on input from the optimization module. The actual sizes then automatically adapt gradually, with VMDKs gaining allocation grabbing SFC blocks from those losing.

With this approach, the speed of cache allocation change automatically reflects the activity level of the VMDKs gaining allocation, again favoring workloads with locality spikes. Such incremental and gradual adjustment also avoids thrashing, where the system oscillates rapidly between two states. Lastly, such lazy eviction maximizes the use of cached data for those VMDKs losing cache allocation.

7 Experimental Evaluation

7.1 Prototype Implementation

We implemented a vCacheShare prototype in the widely used VMware ESXi 5.0 hypervisor.

The trace analysis, CU computation, and optimization modules are implemented as agents in the user world, with ˜2800 lines of C++ code. The modeling and optimization results are persisted via service APIs for the monitoring database currently available on a cluster wide management node.

The rest of vCacheShare is implemented in the kernel, with ~2500 lines of C code. First, to enable runtime cache allocation optimization, an SFC framework was implemented within the hypervisor. All the read IOs below 64KB are intercepted to check if the requested data is cached. The cache itself is managed with LRU, with 4KB blocks. The flash device is used exclusively for caching (including vCacheShare management usage) and cannot be directly accessed by the guest VMs or the hypervisor filesystem layer.

7.2 Test Setup

Unless otherwise noted, our experiments used an HP Proliant DL385 G7 Server, with two AMD Opteron 6128 processors, 16GiB memory, and Intel 400GB PCI-E SSD 910. Local SATA disk drives and an EMC Clariion Storage Array are used to stored VMDKs. The tests used 2-100 VMs, running Windows Server 2008 or Ubuntu Linux 11.04, each assigned a single vCPU, 1GB memory, and a 8GB VMDK.

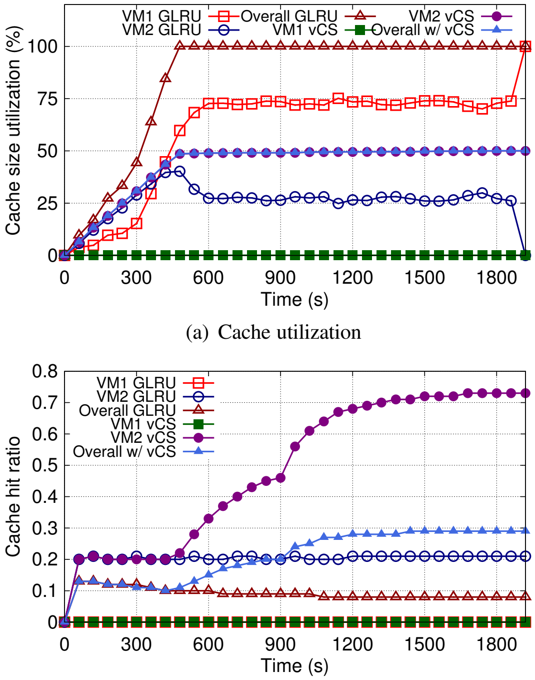
We used the MSR Cambridge traces from SNIA [16]. The traces represent a variety of workloads: user home directories (*usr*), project directories (*proj*), print server (*prn*), hardware monitoring(*hm*), research projects (*rsrch*), web proxy (*prxy*), source control (*src*), web staging(*stg*), terminal server (*ts*), web SQL server (*web*), media server (*mds*), and test web server (*wdev*). They represent IO accesses at the storage disk tier and have accounted for buffer cache as well as application caching effects. This aligns well with vCacheShare's target placement within the hypervisor kernel.

7.3 Result 1: Proof-of-concept Verification

Justifying Cache Partitioning First of all, we demonstrate that explicit, workload-aware cache partitioning is necessary by showing the inadequacy of implicit strategies such as global LRU [7, 20, 21]. In this experiment, we replay two MSR workload traces with a VMware in-house cache simulator with both LRU and vCacheShare replacement algorithms. VM1 runs *src*1_0, which performs a simple data scan, while VM2 plays *prxy*1, with much higher temporal locality. This simulation worked as a proof-of-concept assessment before we set out to implement our vCacheShare prototype, due to the complexity of coding in a real hypervisor.

Figure 5 shows the comparison between using a globally shared LRU cache (GLRU) and vCacheShare (vCS) from a representative segment of the VMs' execution. It plots the VM1, VM2, as well as the overall cache allocation (shown as percentage of the overall cache space occupied) and hit ratio (cumulative from time 0).

This test clearly illustrates the advantage of vCacheShare. With global LRU, the zero-locality VM1 actually grabs *more* cache space than VM2 does, with a hit ratio of near zero all the time. Though LRU automatically favors blocks revisited, the fast scanning behavior of VM1 still prevents VM2 from getting enough cache space. With vCacheShare, instead, VM2 gets sufficient cache space to store its entire working set, allowing the cumulative cache hit ratio to gradually increase as the impact of initial cold misses weakens. VM1, on the other hand, is correctly recognized as of little locality and consequently has hardly any cache space allocation. This avoids the space cost of keeping VM1's blocks in the cache brought in by cold misses, only to be evicted later as in the case of using global LRU. Note that vCacheShare is able to reduce the total cache space usage (50% vs. 100% with global LRU), thereby leaving more space for other VMs, while delivering 21% higher overall hit ratio at the end of the execution segment.



(a) Cache utilization

(b) Cache hit ratio

Figure 5: Cache partitioning effectiveness: vCacheShare vs. global LRU

Hit Ratio Estimation Accuracy We then assess vCacheShare's capability of predicting a workload's cache hit ratio based on its online temporal locality monitoring. Figure 6 depicts the results in terms of *hit ratio estimation error*, which measures the absolute difference between the predicted and actual cache hit ratios, which evolves for each sampling window along the execution timeline (x axis). We replayed week-long SNIA real workload traces in our evaluation. As shown, vCacheShare is able to yield accurate hit ratio estimation in the vast majority of cases, with most data points aggregat-

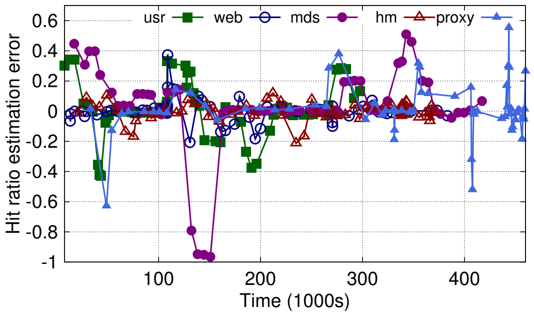


Figure 6: vCacheShare hit ratio estimation accuracy

ing within the [-0.1, 0.1] error range. Meanwhile, there are evident spikes in estimation error, due to fast workload behavior changes. With each spike, vCacheShare is able to correct its large error, though the converging speed depends on the sampling window size as discussed previously. In our analysis of traces collected from a production data center environment, the mean value for the deviation was 0.22 with a standard deviation of 0.4. Fortunately, our reuse intensity metric helps offset the lack of responsiveness to locality changes caused by larger sampling windows.

Necessity of Reuse Intensity (RI) Now we demonstrate that only hit ratio prediction is not enough when the workloads exhibit bursty or intensive accesses within a short time period.

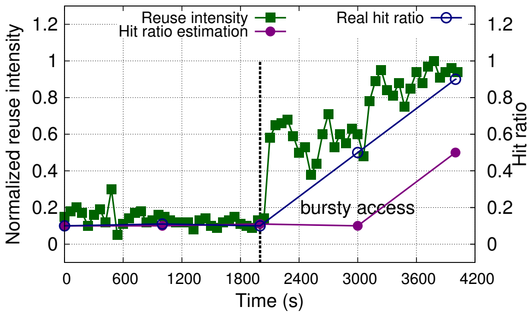
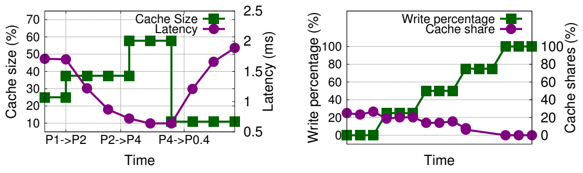


Figure 7: Reuse intensity of MSR trace *web*

Figure 7 shows the RI change (normalized to the largest RI value throughout the execution) of the aforementioned MSR *web* trace. Note the hit ratio prediction is performed on a longer time interval, e.g. several minutes or longer. In contrast, RI is generated more frequently. Figure 7 highlights this by two representative segments of execution concatenated, with small and large hit ratio estimate error respectively. Similar contrast exists for all the traces though. Clearly from the figure, when the hit ratio estimation is close to zero, the RI is close to zero too. However, if there is bursty access between two sampling windows, RI will capture it much more promptly and adapt the cache utility accordingly.

7.4 Result 2: Adaptivity Evaluation

Next, we evaluate vCacheShare's capability of adapting to different policy settings as well as storage backend performance settings. In these experiments, we have 4 VMs running Iometer [22] as a workload generator. In the baseline run, Iometer is set to have 100% random read operations. All the VMs are assigned the same IO priority of 1000 and run the same IO workload, accessing per-VM VMDKs with identical configurations. The expected vCacheShare decision is therefore uniform partitioning across all VMs/VMDKs. The total server flash cache size controlled by vCacheShare is 8GB.



(a) IO shares impact

(b) Read-write Impact

Figure 8: Adaptivity

Varied IO Priorities & Varied Read-Write Ratio: Figure 8(a) shows the effect of varying the IO priority setting on vCacheShare's cache partitioning and resulted IO latency. We change the IO priority of one of the VMDKs along the time line (x axis) to a series of settings: P1=1000, P2=2000, P4=4000, and P0.4=400. Following every priority value change, vCacheShare promptly adjusts this VMDK's cache allocation, while the cache partitioning among the other VMDKs remains uniform. The corresponding latency change follows suite, though it takes longer to stabilize as this VMDK's actual cache footprint expands/shrinks. In the second experiment, the read-write ratio for one of the VMDKs was varied using Iometer. With vCacheShare's bias toward read accesses and read locality, this VMDK sees steady reduction in its cache allocation shares, as shown in Figure 8(b).

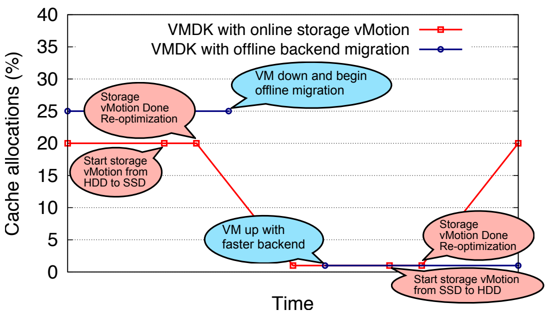


Figure 9: Cache allocation during migration events

Varied Backend Device Latency & Event Handling: In the backend device latency sensitivity test, one VMDK is migrated offline to an flash-backed device. Figure 9 shows the cache allocation for this backend migration scenario. vCacheShare detects a significantly lower service latency from the "victim" VMDK (eliminating the benefit of flash caching) and consequently redistributes nearly all of its cache allocation shares to the other VMDKs.

Figure 9 also shows the behavior of vCacheShare during an online storage vMotion event that moves one VMDK non-disruptively between storage devices. During this transfer, a transient phase where data is sequentially read from the source disk could skew the locality analysis. vCacheShare's event handling design allows it to suspend its analysis till the completion of the vMotion operation. Since vMotion is from disk-backed device to the flash-backed device, the lower IO service time causes a reduction in the cache allocation, which is reversed when the same VMDK is later moved back to HDD.

7.5 Result 3: End-to-end Evaluation

We then evaluate the overall performance of vCacheShare using different workloads.

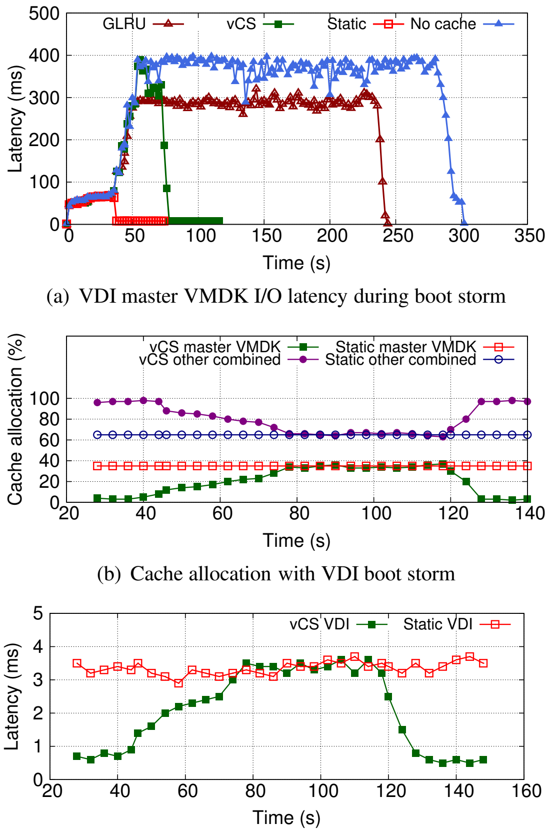
Iometer Test Case: Here we deploy dynamic Iometer benchmark workloads in two VMs (with the same IO priority) sharing SFC. RI is calcuated here with t_w set to 30 seconds. Iometer profiles are initially identical for both VMs but adjusted at run time, producing varied IO working set and intensity. More specifically, we manipulated access locality by (1) shrinking VM1's footprint around time point 480s to increase data reuse level and (2) at 1140s letting VM2 replay its past 300 seconds' access traces, to create an artificial reuse burst.

Figure 10(a) shows the cache allocation changes (except for static allocation, which produces fixed 50% – 50% split). It can be seen that vCacheShare made considerable adjustment to space allocation in the two aforementioned phase of locality change, while global LRU made only minor adaptation during VM2's replay and basically ignored VM1's footprint change.

As a result, vCacheShare brings better overall performance, with Figure 10(b) showing the average access latency averaged over 30s time windows. Compared to GLRU and static allocation strategies, vCacheShare decreased the overall average access latency during the execution segment by 58.1% and 67.4%, respectively.

Virtual Desktop Infrastructure Test Case: VDI has become an important trend in the enterprise. It represents a challenging storage workload, with low average IOPS but transient 10-20X load spikes caused by boot storms or app load storms [23], when all the virtual instances are performing the same task, accessing the base OS image from the *Master VMDK*.

In this experiment, we provision 100 VDI instances based on 4 master images (WinXP, Win2k8 64bit, Win2k8 32bit and Ubuntu 11.04 64bit). Each master VMDK is 8GB, running an in-house VDI workload generator. The total SFC is 80GB and the static cache size applied for each master VMDK is the entire virtual disk size. SFC is managed with vCacheShare, static and global LRU (GLRU) allocation policies. Along with VDI VMs, one additional read intensive Iometer VM (IO access footprint as 80GB) is sharing the same SFC. The RI calculation uses a t_w of 2 seconds. Figure 11 shows the average (among all master VMDKs) latency improvement for master VMDK, the cache allocation as well as Iometer VM's latency during the boot storm process. The performance of a no-SFC configuration ("No cache") is also shown. The boot storm starts at 2 seconds for all experiments and finishes at 300 seconds without caching, 116 seconds for vCacheShare, 74 seconds for static allocation, and 244 seconds for GLRU. The master VMDK has to be brought in from disk-based backend, creating severe queuing delay before the cache is warmed up. This process takes around 38 seconds (till the 40s time point).



(a) VDI master VMDK I/O latency during boot storm

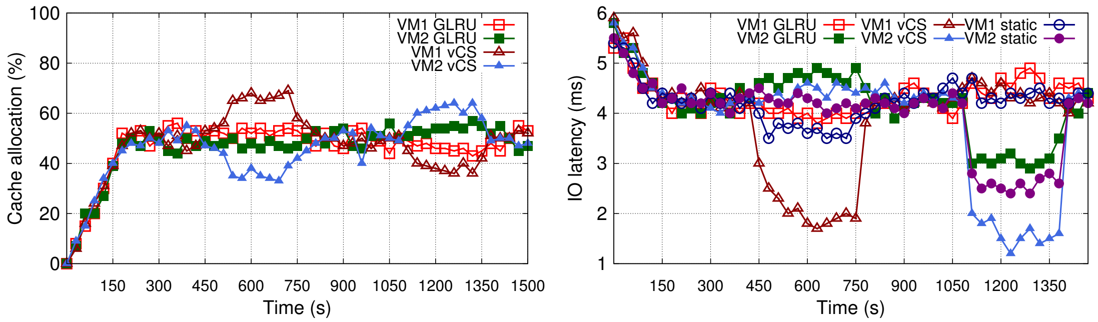
(b) Cache allocation with VDI boot storm

(c) Iometer VM latency

Figure 11: VDI cache allocation and latency

The boot storm lasts over 240 seconds for no cache and GLRU. However, in the vCacheShare setup the load spike is captured quickly by its reuse intensity (*RI*) monitoring. After the master VMDK is cached, the boot storm completes using only 114 seconds, 37.5% and 46.2% of time consumed by no cache and GLRU respectively.

Static allocation starts caching once the boot storm

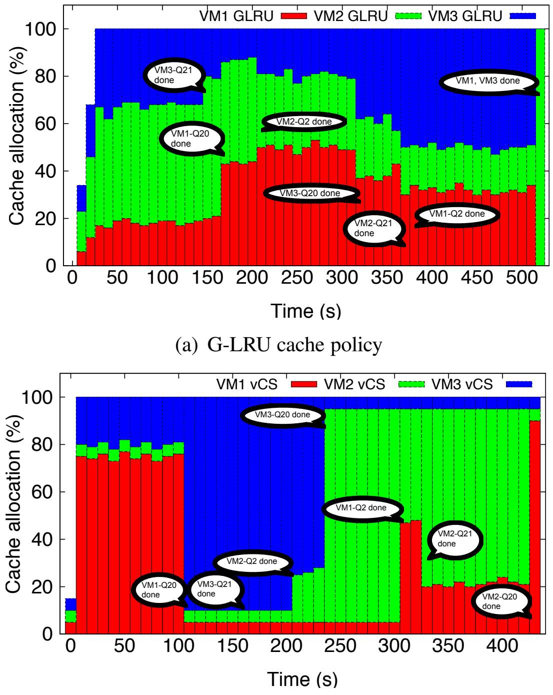


(a) Cache allocation

(b) IO latency

Figure 10: Iometer test results

starts, so it serves boot storm out of SFC the fastest, thus beat vCacheShare during this process. However, it can not reclaim the cache space allocated to master VMDK after the boot storm is over. When there are multiple versions of master images, more and more SFC space needs to be allocated for such one-time use. Worse, static management itself is not able to adaptively adjust cache size for different workloads. From Figure 11(c), although static SFC management shortens the boot storm, it significantly degrades the performance of other VMs before and after boot storm. In our test case, Iometer performs 81% better with vCacheShare compared to static allocation in terms of average latency.



(a) G-LRU cache policy

(b) vCacheShare policy

Figure 12: Cache allocation for TPC-H queries

Data Warehousing Test Case: In this test, we setup our test case using TPC-H [24] (using a scale factor of 3), with Postgres DB configured with default values on an Ubuntu 11.04 VM. A scale factor of 3 is used for the TPC-H database, with Postgres configured with default values. RI here is calculated with t_w set to 10 seconds. TPC-H contains a collection of 22 queries, demonstrating varied data locality patterns. In our experiments we set up three identical VMs, each running the same three OLAP queries (similar to standard TPC-VMS [25] settings), but in different order: VM1 runs queries 20, 2, 21, VM2 runs 2, 21, 20, and VM3 runs query 21, 20, 2. Among them, query 20 has highest locality, 2 has little, and 21 sits in between. Here we omit static partitioning results as its performance is inferior to GLRU.

Results in Figure 12 reveal three major observations. First, vCacheShare makes very different partitioning decisions compared to GLRU, assisted by cache utility modeling to bias toward workloads that can benefit more from caching. E.g., the VM running query 2 (with the lowest locality) actually receives *more* allocation than other VMs when using GLRU, due to its higher IO access rate. vCacheShare correctly detects its lack of data reuse and reduces its allocation to the default reserve level. Second, when there is a workload change (switch between queries), vCacheShare is able to converge to a new relative allocation level faster than GLRU, with its active RI modeling plus hit ratio prediction. Finally, as a result, vCacheShare is able to improve the overall performance by 15.6%, finishing all queries in 430 seconds compared to 510 seconds with GLRU.

8 Related Work

SFC solutions are rapidly gaining adoption, with systems such as EMC VFCache [1], NetApp Mercury [2], as well as schemes developed at Fusion-io [4] and Facebook [26]. To our best knowledge, existing commercial solutions support only static, administrator-defined SFC partitioning or global sharing policies.

Related academic studies have demonstrated that global LRU is not efficient compared to partitioned shared cache for both CPU [7, 20] and disk [21]. Argon [27] partitions the memory cache between different services, providing isolation between the hit rate of each service. The difference with our work is that Argon optimizes the cache hit rate for individual services, while vCacheShare optimizes the overall cache utilization (for aggregate I/O performance). Additionally, unlike vCacheShare, Argon requires administrator involvement each time there are changes in the workload pattern. Janus [28] performs flash allocation optimization for tiered flash-disk storage. The major difference here is that vCacheShare targets SFC systems in block granularity while Janus optimizes file based tiered storage.

The most closely related recent work on flash cache partitioning is S-CAVE [29]. Its optimization is based on runtime working set identification, while vCacheShare explores a different dimension by monitoring *changes* in locality, especially transient bursts in data reuse.

Memory virtualization [30] facilitates transparent sharing of memory among multiple VMs. Recent interest in using flash as extended memory [31, 32] has focused on resolving the access semantics and extending the interfaces beyond block IO, rather than the dynamic space management issues addressed in this paper. Similarly, techniques proposed for CPU cache partitioning [7, 8, 9, 20, 33] target problem settings significantly different from IO caching, which is much more resource (both compute and storage) constrained.

vCacheShare leverages several well-studied concepts in analytical modeling, optimization, and execution. E.g., reuse distance analysis has been used in memory access patterns for decades [14, 34], including cache management policies such as LRU-K [35] and LRFU [36]. Recently, Xiang et al. theoretically proves that hit ratio can be constructed from reuse distance [37], while vCacheShare demonstrates it in practice. Distance-based analysis of temporal and spatial localities has been characterized for file system caching [35, 36]. Low level disk access patterns have been analyzed for uses such as file system or disk layout optimizations [38, 39]. vCacheShare complements existing work by contributing a new approach in locality monitoring based system self-configuration, which may potentially be applied beyond SFC partitioning.

9 Conclusion

In this paper, we presented the motivation, design, and evaluation of vCacheShare, a dynamic, self-adaptive framework for automated server flash cache space allocation in virtualization environments. Through our implementation and experimentation, we confirmed that long observation windows are desirable for accurate cache hit ratio estimation, but may cause slow response to locality spikes and bring higher trace collection/processing overhead. This can be compensated by employing simultaneously short-term locality metrics. Meanwhile, the relationship between observation window size and cache hit ratio estimation accuracy requires further study.

We have also demonstrated that continuous IO access monitoring and analysis is affordable for the purpose of cache space management, even with today's high-VM-density environments.

Acknowledgement

We thank the reviewers for constructive comments that have significantly improved the paper. This work originated from Fei Meng's summer internship at VMware and was supported in part by NSF grants CNS-0915861, CCF-0937690, CCF-0937908, and CNS-1318564.

References

[1] EMC VFCache. `http://www.emc.com/about/news/press/2012/20120206-01.htm`.

[2] Steve Byan, James Lentini, Anshul Madan, and Luis Pabon. Mercury: Host-side Flash Caching for the Data Center. In *MSST*, 2012.

[3] SANRAD VXL Software - Accelerate Virtual Machines with Host Based Flash. `http://www.sanrad.com/VXL/4/1/8`.

[4] Turbo Boost Virtualization. `http://www.fusionio.com/products/ioturbine/`.

[5] FlashSoft Reduces I/O Latency for Increased Server and Storage Performance. `http://www.sandisk.com/products/flashsoft/`.

[6] Proximal Data AutoCache. `http://proximaldata.com/`.

[7] Moinuddin K. Qureshi and Yale N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *MICRO'06*.

[8] J. Chang and G.S. Sohi. Cooperative Cache Partitioning for Chip Multiprocessors. In *ICS'07*.

[9] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *PACT'04*.

[10] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Modeling Workloads and Devices for IO Load Balancing in Virtualized Environments. *SIGMETRICS'09*.

[11] S. Uttamchandani, L. Yin, G. Alvarez, J. Palmer, and G. Agha. Chameleon: A Self-evolving, Fully-adaptive Resource Arbitrator for Storage Systems. *USENIX ATC'05*.

[12] IDC Worldwide Server Virtualization Tracker. `http://www.idc.com/tracker/showproductinfo.jsp?prod_id=39`.

[13] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX ATC'02*, 2001.

[14] C. Ding and Y. Zhong. Predicting Whole-program Locality through Reuse Distance Analysis. In *ACM SIGPLAN Notices 2003*.

[15] 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.

[16] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical Power Management for Enterprise Storage. *Trans. Storage*, November 2008.

[17] Fred Schimscheimer. Server and Storage Sizaing Guide for Windows 7. Technical report, VMware. Inc, 2011.

[18] Simulated Annealing Information. `http://www.taygeta.com/annealing/simanneal.html`.

[19] Hill Climbing. `http://en.wikipedia.org/wiki/Hill_climbing`.

[20] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, September 1992.

[21] Dominique Thiébaut, Harold S. Stone, and Joel L. Wolf. Improving Disk Cache Hit-Ratios Through Cache Partitioning. *IEEE Trans. Comput.*, 41(6):665–676, June 1992.

[22] Iometer: an I/O subsystem measurement and characterization tool for single and clustered systems. `http://www.iometer.org/`.

[23] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The impact of Management Operations on the Virtualized Datacenter. In *ISCA '10*.

[24] TPCH-H. `http://www.tpc.org/tpch/`.

[25] TPC-VMS. `http://www.tpc.org/tpcvms/tpc-vms-2013-1.0.pdf`.

[26] Releasing Flashcache. `http://www.facebook.com/note.php?note_id=388112370932`.

[27] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST '07*.

[28] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, Francois Labelle, Nathan Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *USENIX ATC*, 2013.

[29] Tian Luo, Siyuan Ma, Rubao Lee, Xiaodong Zhang, Deng Liu, and Li Zhou. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *PACT'13*.

[30] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI'02*.

[31] Mohit Saxena and Michael M. Swift. FlashVM: Virtual Memory Management on Flash. In *USENIX ATC'10*.

[32] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *USENIX NSDI'11*.

[33] Nathan Beckmann and Daniel Sanchez. Jigsaw: Scalable Software-defined Caches. In *PACT'13*.

[34] R.L. Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[35] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD '93*.

[36] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies That Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Comput.*, 50(12), December 2001.

[37] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: A Higher Order Theory of Locality. In *ASPLOS '13*.

[38] Sumit Narayan and John A. Chandy. Trace Based Analysis of File System Effects on Disk I/O. In *SPECTS'04*.

[39] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX ATC'07*.

Missive: Fast Application Launch From an Untrusted Buffer Cache

Jon Howell, Jeremy Elson, Bryan Parno, John R. Douceur
Microsoft Research, Redmond, WA

Abstract

The Embassies system [18] turns the web browser model inside out: the client is ultra-minimal, and hence strongly isolates pages and apps; every app carries its own libraries and provides itself OS-like services. A typical Embassies app is 100 MiB of binary code. We have found that the first reaction most people have upon learning of this design is: how can big apps start quickly in such a harsh, mutually-untrusting environment?

The key is the observation that, with appropriate system organization, the performance enhancements of a shared buffer cache can be supplied by an untrusted component. The benefits of sharing depend on availability of commonality; this paper measures a hundred diverse applications to show that applications indeed exhibit sufficient commonality to enable fast start, reducing startup data from 64MiB to 1MiB. Exploiting that commonality requires careful packaging and appropriate application of conventional deduplication and incremental start techniques. These enable an untrusted client-side cache to rapidly assemble an app image and transfer it—via IP—to the bootstrapping process. The result is proof that big apps really can start in a few hundred milliseconds from a shared but *untrusted* buffer cache.

1 Introduction

When a user installs a new desktop application, he accepts the risk that the new app may compromise any other app he uses. In contrast, web sites he visits are responsible for managing their own servers; a visit to a new site doesn't present a threat to the servers that run other sites he uses. A site manager is better equipped than her users to make security decisions about her server, and the server's isolation gives her the autonomy to effect those decisions.

This benefit should accrue to the web as a whole, except that the client side is bloated and vulnerable; thus clicking a web link can be as risky as installing a desktop app. The Embassies project [18] proposed refactoring the web client interface to isolate client-side apps as effectively as servers are isolated in multitenant data centers, so that the site manager becomes autonomously responsible for her client code, too. We call this model the "pico-datacenter" – the client becomes a hosting site for mutually distrustful applications, providing no semantics other than a VM-like container, IP and the thinnest UI interface (each app paints raw pixels on its part of screen).

The Embassies design aims to mimic the relationships among software components found in a shared data center: each vendor enjoys strong isolation, retaining autonomy even as it communicates with other vendors. This isolation promises to protect Embassies from the bloat that afflicts prior client models; but it demands a truly minimal host.

Unlike Embassies' pure shared-nothing model, though, existing web clients extract substantial performance benefits from sharing. The host operating system's buffer cache and the browser's HTTP object cache share content across sites. The lumbering 100 MiB browser process itself is shared, since it need not restart for each new site the user visits. Many people's first reaction to the Embassies proposal is alarm at the idea of shipping such big apps around; surely it must lead to unbearably slow app launch times?

Surprisingly, such big apps can be started nearly as quickly as a conventional web page. It is one thing to make an abstract argument that it should be possible; the aim of this paper is to decisively demonstrate so. This paper shows that ideal isolation does not funamentally conflict with good application-launch performance.

We construct a content cache that is untrusted (as untrusted as a random neighboring tenant in a shared datacenter), and yet enables mutually distrustful sites to share content and reap the benefit of fast app launch, while using end-to-end cryptographic checks to protect their own integrity. Essentially, we show that the OS buffer cache and browser object cache can be evicted from the trusted computing base (TCB) and replaced with an untrusted cache that delivers similar performance benefits.

For the untrusted cache to be effective, there must be commonality to exploit; we must demonstrate the performance equivalent of many applications sharing a common browser infrastructure. This paper

- demonstrates that a hundred diverse applications exhibit great commonality, enabling efficient transfers and fast launches
- shows an integrated pipeline that packages, transfers, caches, and launches large application images in a secure manner.
- characterizes the sensitivity of performance to

pipeline parameters, and

- demonstrates hot- and warm- app launch times comparable to that of a conventional OS buffer cache and shared library mechanisms (in which apps are mutually trusting).

Missive is best motivated by the extreme minimality of Embassies [18], but it applies more broadly. Other client app delivery systems such as Tahoma [10], Xax [17], Native Client [43], and Drawbridge [33] ship VMs or large binary programs, and an untrusted cache would reduce their TCBs. VM images in any context are big, and launch times can be slow [4, 24]. Fast launch is particularly relevant for security applications that spawn a VM per user [32] or per connection attempt [41].

2 Context

We focus on Missive's applicability to the Embassies client application environment [18], since it takes host minimality to the extreme, making a shared cache particularly challenging.

2.1 Embassies Overview

With Embassies, apps are strongly isolated, communicating with other apps and with the outside world only via IP. The intent is to create an environment analogous to the server app environment, where each vendor is completely isolated from other vendors and exercises full control over its own software stack: If a server app is compromised, it is because the vendor chose a poor library, misconfigured a firewall, or failed to patch its software. No careless or ignorant user decision can be blamed. By analogy, on an Embassies client, the user's decision to open a new app cannot compromise any other app on the client, since the apps are as isolated as two tenants on a hosting server. The simple communication semantics of IP make it clear how a vendor protects itself: if a vendor's software selection and administration can protect its server-side software from attacks arriving over the Internet, then the same follows for its Embassies client software.

The Embassies model deviates from the server-side model in a few respects. For example, it offers apps raw, pixel-level access to the display for interactivity. However, the most important distinction is the workload; typical server software multiplexes many users over one installation of long-lived code and database. In contrast, at the client, we expect the user to frequently switch context between apps and to often launch altogether new apps (analogous to clicking on links in a conventional web browser). Worse, these apps are likely to be large: Rather than a skinny JavaScript atop a big shared browser or a small executable atop a dozen shared libraries, each app is more like a standalone Virtual Machine (VM) image. However, in pursuit of strong isolation, the Embassies client platform aims for minimality, which obstructs conventional performance optimizations that tightly couple sharing of libraries and caches.

Indeed, the Embassies client omits facilities for a buffer cache or wide area transfer (MIME, HTTP, or even TCP). Missive fills that gap, showing how *mutually-untrusting* applications can cooperate to exploit the sharing opportunities that lead to good performance.

2.2 Embassies App Start

Figure 1 shows how an app starts on an Embassies client. First, some *invoking* app A, perhaps one in which the user has clicked a link, identifies a public key that represents the target app B; it also fetches B's signed boot block. It is app A's responsibility (not the client kernel's) to verify that it has found the correct principal (here it uses DNSSEC).

Second, in steps 2 through 5, the kernel receives the signed boot block, checks the signature, associates a fresh process with the signature's public key (B), and begins executing the boot block in the new process. This is the only phase whose shape is imposed by the Embassies TCB.

Third, in steps 7 through 10, the untrusted cache gathers the metadata and data required to assemble the image for App B. None of these interactions require integrity other than to avoid wasting the cache's time.

Finally, the cache sends the entire image in a single IPv6 jumbo packet into App B's process. App B then verifies the image's integrity, for example, by checking the image's hash against a hash value included in the boot block. Finally, App B transfers control to the code in the image.

For communication between apps and to the greater Internet, the client kernel provides only an untrusted IP channel. The mapping of name to app key, fetching of boot block, and fetching of image content are all built from the IP primitive, making it easy to reason about isolation.

The client kernel provides no storage; instead, it is assumed that some anonymous vendor (e.g., Seagate) provides an untrusted, insecure storage app. The client kernel does provide each application with a single secret specific to the host and the app's identity:

$$K_{app} = \text{PRF}_{K_{host}}(\text{ID}_{app}).$$

That secret is only available to processes started from a suitably-signed boot block. The secret enables the app to convert untrusted storage into secure storage via encryption and cryptographic integrity checks, while requiring the client kernel to store no per-app state.

These are all of the primitives Embassies offers for the app-start process. Missive's mission is to provide high performance app starts from only these primitives. One

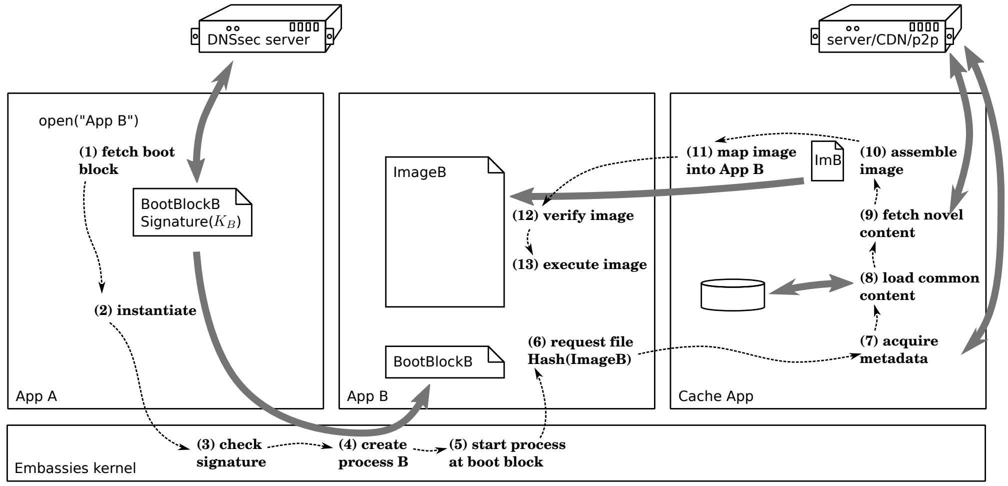


Figure 1: **App start process.** *To launch a new process for App B, App A fetches a signed boot block for B, the kernel verifies the signature, the cache assembles the required full application image, and sends it to App B to execute. Dashed lines show control flow. Heavy grey arrows show data flow. All data flows, other than the boot block passing through the kernel, are via IP.*

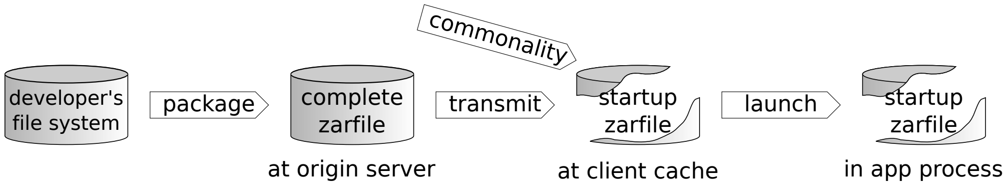


Figure 2: **System diagram.** *The developer's files that comprise the app image are packaged into a "zarfile". The client cache fetches the part of the zarfile required for app start, and the zarfile is delivered into an isolated process to launch the app.*

naïve approach would appoint a distinguished app vendor to supply the trusted cache, but then to enjoy the performance benefit of a shared cache, apps must trust the cache vendor; it becomes an implicit part of the TCB. Instead, Missive's architecture lets every app exploit a single shared cache, without trusting that cache.[1]

3 System Design

Missive comprises three steps (Figure 2): The *packaging* step collects binary libraries and data files from a developer's machine into an image, called a *zarfile*. Files are placed in the zarfile to expose sharing opportunities. The *transmission* protocol transmits zarfiles across the network; it is designed to minimize round trips, exploit commonality to minimize bandwidth, and enable incremental access. The *launch* procedure transfers a zarfile from an untrusted cache into the booting app's process, with a focus on minimizing the startup latency.

3.1 Packaging

An Embassies app is completely specified by its vendor. Thus, the vendor can configure the app on a development machine using any available framework or tool. She might install required framework components, such as a Python math library; she might carefully select a specific version of a subpackage (such as an audio rendering library); and she might even hand-patch a component or configuration file to fix a security vulnerability.

Once all of the components are in place, the vendor runs the packaging tool, which enumerates the set of files that comprise the app, including the app executable, data files, libraries, and library OS components [17, 20, 33]. This is the *complete* application image. The tool also captures a dynamic run of the application, identifying the subset of the complete image necessary to bring the app to a usable interactive state; this is the *startup* set. The startup set is captured at sub-file granularity so that it skips chaff such as symbols.

By identifying the startup set, Missive enables the developer to reduce the size (and increase the speed) of the initial app transfer. After app launch, the remaining components may be fetched from the complete image on demand, or preemptively in the background, so that they will be available when the client is disconnected.

Missive's zarfile is a simple tar-like format. It specifies a master index, string and data-chunk lookup tables, file `stat` metadata, and file contents.

Below, we elaborate on the challenges involved in im-

[1] Missive does not prevent side channels: by probing the cache's response time, one app can learn about content accessed by others.

age capture and ensuring zarfile stability. The capture process also honors memory layout constraints designed for fast app start as described in §3.3.3.

3.1.1 Image Capture

Some tuning is required to extract a minimally-sized image from a conventional POSIX development system. For example, we found that the Gnome system-wide icon cache may be 50 MB, but a single app may access only a few kilobytes of icons from it; our packaging tool strips the icon cache apart to avoid the waste. Similar techniques could be applied (although we have not yet done so) to strip unneeded code from shared libraries. (On the other hand, leaving libraries untuned may enhance commonality; see §4.)

3.1.2 Image Stability

As §4 discusses in detail, a critical component of Missive's good performance is detecting common content shared between indifferent apps. To facilitate this detection, Missive's packaging tool is aware of the block size used during transmission (§3.2), and it strives to ensure that small changes in file selection, file content, or file length will produce zarfiles in which most blocks have the same content and location.

Block Content Stability. In typical file-size distributions almost all files are small files, but a few large files comprise almost all the bytes [2, 12, 36], and our file set is no exception (Figure 3). The tail of tiny files makes it impractical to give every small file its own block; padding would expand the image by $2 - 10\times$ (§5.2), wasting too much physical memory.

Instead, Missive's packager aligns large files—those bigger than a block—on block boundaries to maximize commonality detection, and it uses small files to fill in the gaps left at the end of large files. While some extra space still remains, in practice, the overhead of padding in a zarfile is generally below 2% (§5.2). In wide-area transit, the wasted bytes are compressed away, while the effect on disk is negligible. In memory, the bytes are moved cheaply by reference, so overall, the layout has little performance penalty.

The benefit of this layout is that it makes it likely that, if two zarfiles share many large input files, then they contain proportionally many identical blocks. Furthermore, a change in a small file affects only the block whose tail it occupies. Thus, for each file different between two images, the zarfiles differ by $\lceil \frac{file\_len}{block\_size} \rceil$ blocks.

In summary, the packaging tool ensures that two similar zarfiles share almost all of their aligned blocks. Therefore, launching an app similar to one already cached requires transmitting bytes proportional only to the amount the images differ.

Block Position Stability. The block-aligned layout policy described above is close to what we want, but it leaves three problems. First, a change in the size of a big file or the set of small files may cascade, changing all of the tail-gap assignments for the small files. Second, perturbing the file order perturbs all of the Merkle hashes in the transmission phase (§3.2), foiling opportunities to share Merkle subtrees.

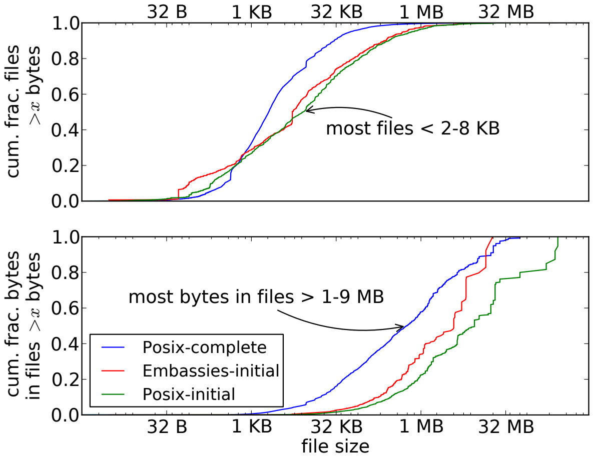


Figure 3: **File size distribution.** *All of our experimental data sets (details in §4.1) obey the typical distribution: almost all files are small, and almost all bytes belong to the few big files.*

Third, and most importantly, since we strive so enthusiastically for minimality, we cannot assume the kernel supports a `gather-send` operation. Yet we still wish to extract maximum performance. Absent gather, if two zarfiles share most of their blocks, but those blocks are reordered, construction will require the cache to copy all of the blocks into the correct offset in the outgoing message. For our parameters, the copying alone can add 100–150 ms to startup latency. When we provide position stability, the cache exploits it in the warm case by assembling the first zarfile in a buffer with some slack memory at the end. Then, when a request for the second zarfile arrives, the cache need only patch the blocks that differ from the first zarfile.

To foster position stability, we refine Missive's placement algorithm to produce zarfiles that not only share blocks with common content, but whose common blocks will appear at common offsets. Let t be the total size of the input files, i.e., the minimum size of the output. Let u be the nearest power of two greater than t, and l be the nearest power of two less than t. Pick a random $seed$, and hash each input file f along with $seed$, producing $h_f = hash(f||seed)$. If $h_f \bmod u < t$, then the file's preferred placement is $h_f \bmod u$; otherwise, the preferred placement is $h_f \bmod l$. We truncate all preferred placements to block boundaries to produce, for each file, a preferred block-aligned location in the range $[0, t)$. For each file, we evaluate the placement expression with ten seeds to produce a prioritized list of preferred locations.

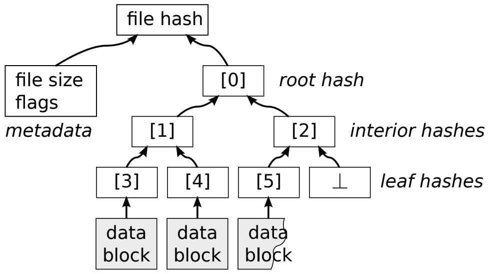


Figure 4: **Image file protocol representation.** *Each arrow represents an input to a hash operation. The Missive transmission protocol represents a zarfile with a conventional Merkle tree, plus the metadata required to define the tree's dimensions. After learning the metadata, the receiver can query hashes and data blocks in any order, and verify them incrementally.*

The input files are placed into the zarfile greedily, by descending file size. If a file collides with a prior placement, we try its lower-priority placements. If no preferred placements work, the file is dumped into a leftover bucket. After every file's preferred placements have been attempted, the leftover files are placed into gaps or concatenated to the end of the file.

Intuitively, the algorithm ensures that: (a) A change to a small file will, due to the greedy placement order, make small changes to the overall zarfile, since most bytes have already been placed by the time the change impacts the algorithm. (b) A change to a large file perturbs the algorithm early, but only affects that file and those later files whose placement depend on a collision created or eliminated by the change. This contrasts with the basic greedy algorithm where any change affects *every* later placement decision. Conversely, a change to a large file that preserves its length will not perturb the basic algorithm, but will perturb the position-stable algorithm.

3.2 Transmission

Once the zarfile is defined, it must be fetched to the client; this occurs using well-understood techniques: To preserve the integrity of the vendor's app image specification, zarfiles are specified by content. To exploit commonality, the content is specified by hashing blocks; blocks already cached at the receiver because they're common with other apps do not require transmission. Finally, to enable the app to fetch subsets of the zarfile (such as fetching the startup set from inside the complete zarfile), the block hashes are arranged into a Merkle tree [27]. The file is self-certified [16] by the *file hash*, i.e., a hash of the Merkle tree root and the file metadata (Figure 4). The metadata consists of the file length and a flag indicating whether the file contents should be interpreted as a directory.

The Missive transmission protocol is packet-based and incrementally self-verifying (Figure 5). In the first round, the receiver asks for a file by its file hash, and receives the file metadata; because of the structure of the file hash, it can immediately verify the reply. The receiver determines the data flow, so in each later round, the receiver may request any subset of the Merkle node hashes by their tree indices, and may request any byte range of the file contents.

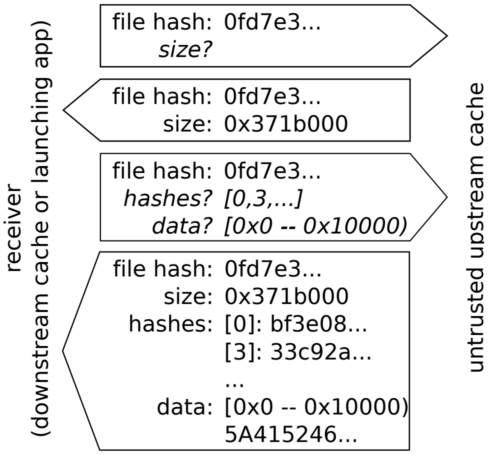


Figure 5: **Transmission protocol.** *The meaning of each reply packet is independent of the query that spawned it, and each reply contains enough information to verify its contents.*

A bandwidth-constrained receiver may learn the tree one level at a time to avoid fetching even the Merkle hashes of subtrees it already knows; each layer can be incrementally verified. A receiver desiring to minimize round trips will instead ask for all the required leaf hashes in one step, plus any Merkle siblings required to compute all the required interior nodes. Section 5.1 analyzes the trade-off. Our implementation uses the latter algorithm.

Once the leaf hashes are known, the receiver can fetch actual file data, verifying that the data is sane at block granularity. We use a 4 KiB block size, which has reasonable overheads (§5.2).

Because the protocol is receiver-directed, it adapts to diverse scenarios. The receiver chooses which and how much hash and file data arrive in each reply, and hence adapts to networks with varying MTU. It detects integrity failures immediately for metadata and Merkle hashes, and after receiving each block's worth of file data, so it can quickly reject faulty senders. The receiver selects whether to compress file data, based on whether network bandwidth or receiver CPU is scarcer. When an app receives transmissions from a cache on the same host, we use UDP to keep the boot block tiny; in a cache receiving transmissions across the network we use TCP to tolerate latency and congestion.

3.3 Launch

When a new application starts, it consists of a tiny binary boot block running in an isolated process (§2.2).

The boot block contacts a cache on the local machine and asks for its application image by file hash, requesting the required range of bytes. Since the cache is not trusted, the boot block can locate the cache by broadcast. Thus the boot block is quite simple and only understands a subset of the Missive protocol; it leaves the work of fetching the image to the cache, though it may provide a URL or other hint as to where the image may be found.

Launch is optimized to minimize latency, which comes from *data transfer*, *verification time*, and *application start overheads*.

3.3.1 Data Transfer Latency

In a conventional buffer cache, data transfer is very fast: the application names a few dozen files to map, and the OS page-remaps those files from the buffer cache to the process' address space. In the context of Embassies, Missive's untrusted cache achieves a similar effect by transmitting the entire image in a single IPv6 jumbo frame. The Embassies client kernel implements large local transmissions with page remapping, while preserving IP's predictable by-value semantics.

The untrusted cache can only exploit this performance boost if it can prepare the image for transmission efficiently (§3.1.2).

3.3.2 Verification Latency

Since the buffer cache is untrusted, the boot block must verify the image the cache provides before it starts executing the image. This verification is on the critical path. The boot block's contents are public, so it initially has no secret key with which to perform verification. Thus, the boot block includes a hash value, e.g., from SHA-256, to verify the integrity of the image it receives from the cache.

Unfortunately, secure hashes are costly. To reduce the cost of verification in the common case, the boot block substitutes the computation of a hash with a Message Authentication Code (MAC), a faster message summary that requires the sender and receiver share a secret. Initially, the boot block does not have such a secret, so on its first execution, it verifies the image based on its hash. Once the hash verifies, the boot block computes a MAC over the same data, using K_{app}, the app-specific secret key provided by the client kernel (§2.2). The boot block stores the MAC in untrusted storage (e.g., with the cache), since the use of a secret key makes the MAC neither private nor subject to integrity attack by the untrusted store. Standard techniques (not implemented here) can prevent attacks on data freshness [25, 31].

On subsequent starts, the boot block queries the cache for the MAC it previously computed. If the MAC is present, the boot block verifies the image's integrity with the MAC, skipping the hash altogether, resulting in a faster startup. We use VMAC [23], a MAC ten times faster than SHA-1. On our experimental hardware (§5), SHA-1 costs 3.9 ms/MiB ($\sigma = 0.1\%$), whereas VMAC is 0.29 ms/MiB ($\sigma = 1.0\%$).

We aim to reduce user-perceived startup latency, and both hash and MAC computations are embarrassingly parallelizable. Thus our boot block exploits all available cores to trade a wide burst of computation for reduced latency.

Another way to reduce the verification latency is to overlap it with later steps: the host could provide additional primitives to allow speculative execution [29] while the verification process continues (not implemented here). Embassies discards this option because it adds additional host complexity.

3.3.3 App Start Latency

Once the image has been transferred and verified, the app begins executing. Several factors affect how quickly the app is ready for user interaction.

Page Alignment. In our implementation, the bulk of the image consists of shared libraries. Libraries expect to load at 4 KiB memory-page boundaries. Missive's block alignment policy ensures page alignment for large files. When a boot block requests the zarfile from the cache, it specifies a padding header that compensates for the IP header and host packet buffer offsets, making the first byte of the zarfile fall on a page boundary. Correcting these alignment issues forestalls runtime memcpy operations that otherwise impair startup latency.

ELF Section Layout. The ELF standard adds additional complications: ELF-format files have a non-trivial mapping between on-disk structure and in-memory structure. Typically, an ELF file has a large text segment, then a smaller initialized data segment that is expected to appear at an offset in memory different than its offset from the text segment in the file. The ELF file also specifies an uninitialized data (bss) memory region with no corresponding data in the file, as well as file regions (such as debugging symbols) that are not mapped into memory.

Missive addresses this complexity as follows: at image capture time, it records how regions of the library file are mapped into memory. The text segment is recorded in the zarfile in an oversized region adequate to hold the final in-memory layout. The data segment is recorded in a separate region. At runtime, the data segment is copied into place at the appropriate offset, and the bss segment is zeroed. This arrangement lets the library run directly from the launched image, eliminating the bulk of memory copy operations. The cost of the empty space in the image is tolerable, as zero-filled regions compress nicely for wide-area transfer.

4 Image Commonality

To deliver on the promise of fast launch of big, independent applications, Missive assumes that most applications actually share a fair amount of common infrastructure. Broadly, we conjecture that most apps will exhibit commonality with at least some other apps, because there will be only a few popular app-building frameworks. Over time, some will fork and others will ebb. This evolution will look less like today's Web client, where standards make it difficult to fork away from HTML and JavaScript, and more like frameworks on servers, where Django and Rails evolve competitively. This intuition does suggest increased variability, but not unbounded schisms.

4.1 Characteristics of Our Data Sets

We evaluate this conjectured application commonality by examining the commonality within three data sets drawn from two application populations.

First, we study a population of 100 interactive desktop applications from Ubuntu Linux 12.04. We selected them using Linux "best-of" application lists [15, 42] representing a wide variety of domains (e.g., web browsing, vector illustration, word processing, video editing, music composition, software development, chemical analysis) built using various languages (C, C++, Java, Mono, Python, Perl, Tcl) and GUI frameworks (Gnome, KDE, Qt, Tk, Swing).

From this population, we first constructed the **POSIX-complete** data set: for each application, we constructed a zarfile containing every file the application might touch while running. We used Ubuntu's package management system to find this list: we queried the package manager for all packages that are dependencies of the application's base package. (Such dependencies are declared manually by Ubuntu's package maintainers.) We then recursively found sub-dependencies, eventually enumerating the entire dependency tree for each application. Each application's zarfile contained the union of all files in all packages in its dependency tree.

The second data set, **POSIX-startup**, is drawn from the same population. However, instead of the complete zarfiles, we measured only the portion that is accessed while the app launches to the point of interactivity, as captured by `strace`. This provides a more accurate picture of the time a user might be expected to wait to use an app after launching it.

The third data set, **Embassies-startup**, consists of eight apps we adapted from the POSIX world to run in Embassies to validate that the minimal client kernel really can support rich apps. These include Midori/Webkit (an HTML renderer), Abiword (a word processor), Gnumeric (a spreadsheet), Gimp (a raster image editor, like Photoshop), Inkscape (a vector image editor, like Illustrator), Marble (an interactive globe, like Google Earth), Gnucash (an accounting app, like Quicken), and Hyperoid (a video game). These are the types of real rich applications we would like to see deployed in the Embassies model. These apps use some common and some distinct components: most use X windows as a rasterizer, for example, but some use the Qt graphical toolkit, while others use Gtk. Porting the apps to Embassies entails packaging them into images that encompass executable libraries and runtime data.

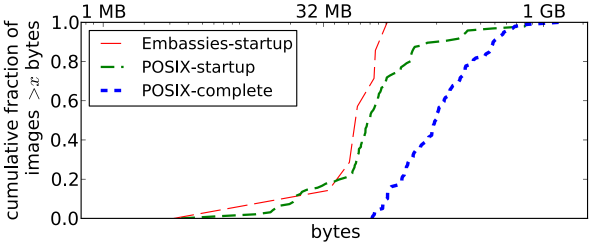


Figure 6: **Zarfile size distribution.** *The distribution of image sizes used to test commonality across images. The -startup sets have medians of 54 and 66 MiB. The complete set images range 68 MiB–1.0 GiB.*

While the Embassies-startup population loses fidelity because it is much smaller than the POSIX apps, it more accurately represents the anticipated ecosystem in that each app image is a real binary that launches in Embassies. The POSIX data sets are less accurate; for example, they omit about 10 MiB of functionality associated with the Embassies POSIX emulation, TCP stack, and X rasterizer.

The agreement between measurements of the POSIX-startup and the Embassies-startup sets suggests that the POSIX-startup set is a reasonable approximation of what the apps would look like if ported to Embassies, and hence we can use this larger set of apps to evaluate such a world. The introduction of the POSIX-complete set lets us reason about the cost of transmitting complete app images for offline use.

Figure 6 shows the distribution of image sizes in each data set.

4.2 Commonality Measurements

To measure commonality, we simulated the transmission of each data set to a client machine assuming that some apps are already cached there. Our hypothesis is that applications share enough common infrastructure (and, thus, common zarfile blocks) that installation of a new application will require significantly less data transfer when other applications are already cached. The block size was 4 KiB.

The results for POSIX-complete are shown in the CDFs in Figure 7a. The bottom curve shows the worst case: transfer of a single zarfile to an empty cache. It is

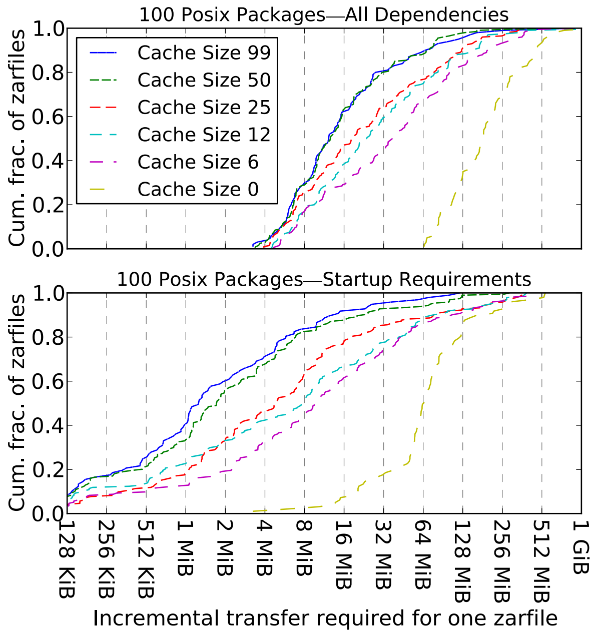


Figure 7: **Content commonality across zarfiles generated for Posix applications.** *Cumulative distribution function of the required transfer size to install the $n+1$st zarfile on a Missive system where n zarfiles are already cached. The block size is 4 KiB. 7a (top) shows complete applications with all dependencies. With 100 apps, the cache reduces the median transfer size from 181.6 MiB to 12.2 MiB. Little incremental benefit is gained beyond 50 cached zarfiles. 7b (bottom) shows results for the portions of the zarfiles required for application start. The cache reduces median transfer size by nearly 98%; typical apps in the startup set exhibit more commonality than the complete zarfiles.*

nearly equivalent to the zarfile size distribution (modulo duplicated blocks), ranging from 64.8 MiB to 1.3 GiB (median 181.6 MiB). The top curve shows the best case: transfer of a single zarfile when all other 99 zarfiles are cached. The number of unique bytes requiring transfer was reduced to an average of only 6.7% of the original, to a median of 12.2 MiB.

The middle curves in Figure 7a show intermediate cases when some (n = 6, 12, 25, and 50) of the 100 zarfiles are cached before the transfer of one additional zarfile. In each simulated transfer, we first populated the cache with n zarfiles selected uniformly at random from the $\binom{100}{n}$ possibilities, then selected one to transfer randomly from among the $100-n$ that remained. The simulation suggests virtually all of the cache's benefit is realized with 50 zarfiles cached.

Figure 7b shows the same experiment performed on the POSIX-startup data set. These zarfile subsets were about 36% the size of the full zarfiles, ranging in size from 3.2 MiB to 543.1 MiB (median 65.6 MiB). But the reduced set isn't just smaller: it also exhibits far more commonality between applications. When all but one were cached, retrieval of the final zarfile required a transfer of between 8.0 KiB and 121.3 MiB (median 1.3 MiB)—a reduction to just 2% of the median transfer with a cold cache. To give these numbers context, a typical app that had not been previously installed could start in less than two seconds on a 6 Mbps cable connection.

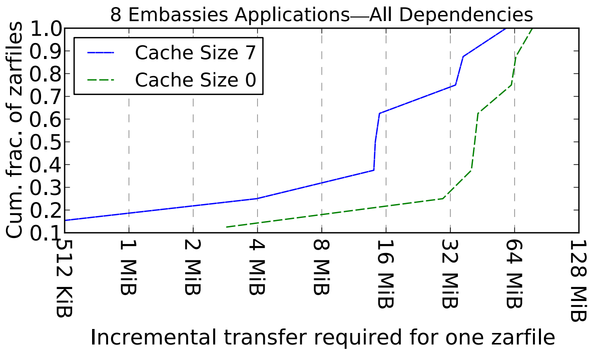


Figure 8: **Content commonality across zarfiles generated for Embassies applications.** *Cumulative distribution function of the required transfer size to install the $n+1$st zarfile on a Missive system where n zarfiles are already cached.*

Figure 8 shows the same analysis for the 8 apps in the Embassies-startup data set. With seven apps cached, transfer time for installation of the eighth was reduced to about 34% of its size. This is roughly comparable to the efficacy of caching for Posix apps with a cache that size. This suggests that in a larger Embassies ecosystem, efficacy of caching will approach the 98% seen in our Posix study.

This simulation does have inaccuracies. It overestimates the available commonality in a real Missive ecosystem because all the applications we tested are from the Posix world, none from Windows or Mac. In addition, where two of our apps share a framework, they use the same version. However, it also underestimates the opportunity for efficient transfer: in an ecosystem that exploits Missive, libraries and frameworks may be repackaged to make their components easier to share; here, no such optimization has been performed. The experiment also understates the benefits of apps that share nearly identical stacks, such as multiple apps built on an HTML renderer.

4.3 Block Size Selection

Block size is an important parameter in Missive. Larger blocks produce less metadata, while smaller blocks might expose more commonality across zarfiles, depending on the distribution of the variations. Figure 9 shows our analysis of Missive's sensitivity to block size for the Posix-complete dataset. We filled the cache with 99 of the zarfiles and simulated the transfer time required

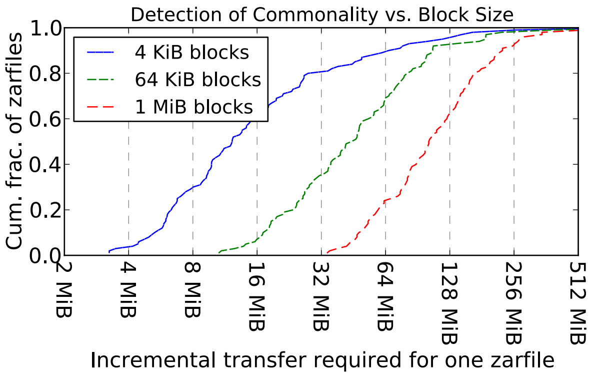


Figure 9: **Sensitivity of commonality-discovery to block size.** *Cumulative distribution function of the required transfer size to install the 100th zarfile on a Missive system where the other 99 are already cached. The size of the blocks hashed is varied. Computing hashes for smaller blocks results in significantly more discovery of commonality across zarfiles and much smaller transfers. Simulation uses the POSIX-complete dataset.*

to install the 100th. With a block size of 1 MiB, that final zarfile required a transfer of between 34.0 MiB and 1.1 GiB (median 102.5 MiB). 64 KiB blocks reduced the median transfer size to 45.3 MiB and 4 KiB blocks reduced it further to 12.2 MiB. The increase in metadata is clearly worth reducing the block size to 4 KiB.

4.4 App Patching

One important special case of commonality is patching: replacing an image already present at the client with a similar one. Note that patching is a domain-specific application of compression, and hence is amenable to specialized optimization. Chrome's Courgette tool produces binary patches ten times smaller than a structure-oblivious binary diff [1].

Missive is designed to extract commonality implicitly across apps, without the receiver identifying a source version, but such specialized tools layer nicely on top of Missive: If an app wants to patch itself, it can identify a previous version against which the patch should be applied, and which is available locally. In that case, a specialized tool like Courgette can be executed either by the app or by the untrusted cache to generate new content from an efficient patch and supply it to the shared cache, ready for future app launches.

5 Evaluation

We analyze the choice of Merkle degree, evaluate the overheads due to Missive's packaging strategy, and measure the overall performance on app startup time. All measurements were collected on an HP z420 workstation with a four core 3.6GHz Intel Xeon E5-1620 CPU and 4GB of RAM.

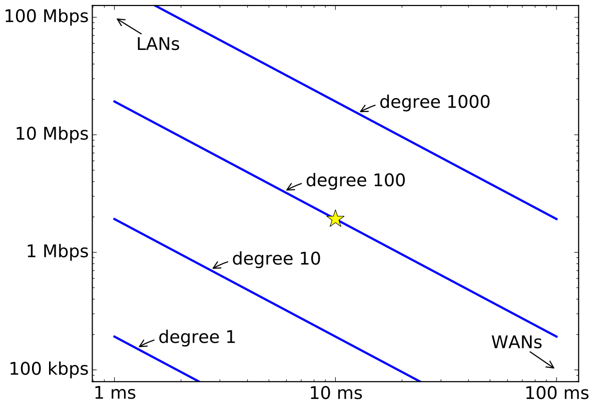


Figure 10: **Ideal Merkle tree degree is a function of bandwidth-delay product.** *On networks with high bandwidth-delay, the receiver might as well request many intermediate hashes rather than spend an RTT hoping to reuse the hashes of an existing subtree.*

5.1 Selection of Merkle Tree Degree

Merkle trees enable a receiver to verify and begin using a partial image before the entire image is transferred and verified [3]. A smaller benefit is that the receiver can skip gathering a subtree of Merkle hashes if the root of that subtree is already cached.

A small-degree tree exposes more such opportunities, but those bandwidth savings come at the cost of round trips. Thus the ideal tree degree is determined by the bandwidth-delay product [21] of the network transport (Figure 10). For example, on a 1.92 Mbps, 10 ms connection (star), receiving 100 192-bit hash records is as cheap as an RTT, thus the optimal tree degree is 100.

We configured our Merkle tree with degree two because it was easy. This value only makes sense for slow networks, but receivers on faster networks can emulate higher degree by requesting multiple layers in each RTT.

5.2 Overheads

The image file format is structured to create opportunities for commonality exploitation, but that structure can introduce overheads. The most important overhead is the alignment-inducing padding, but in practice, our measurements show that it remains below a few percent (Figure 11).

The padding overhead is affected by the choice of block size (Figure 12): larger blocks incur more padding, but increasing the block size also marks more files as “small”, packing them into the tail of partially-used blocks. It turns out padding is worst at 64 KiB: smaller blocks generate less padding per “large” file, and larger blocks reduce the number of “large” files that generate padding. Regardless of block size, the padding never becomes a significant overhead.

Packing small files into block tails hides opportunities

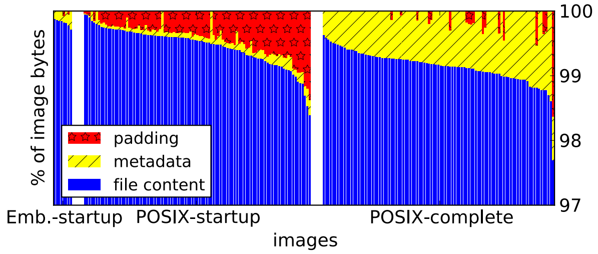


Figure 11: **Image file format overheads run 1-2%.** *The image file format incurs about a percent overhead for metadata such as file stat metadata and a name index. Padding varies depending on the distribution of file lengths, but it remains below a few percent. Note the y axis begins at 97%. Images constructed with 4 KiB blocks.*

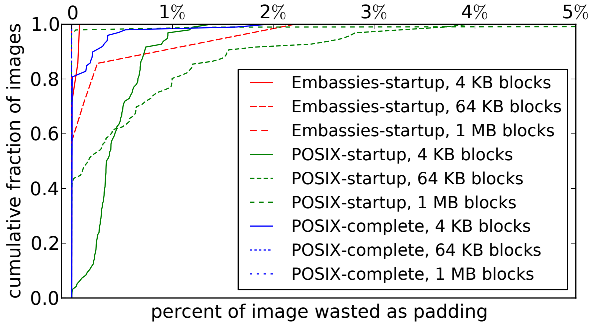


Figure 12: **Padding overhead** *is insensitive to block size.*

for sharing; is it necessary? Yes: although there are few bytes in small files (Figure 3), giving each file its own block introduces considerable overhead that grows with block size (Figure 13).

5.3 The Bottom Line: Startup Latency

Ultimately, we aim to demonstrate that, by judiciously coupling image transmission and buffer cache, Missive achieves interactive performance without a trusted file system or buffer cache infrastructure. To that end, we measure end-to-end network transmission and application launch times.

5.3.1 Launch Time

Besides fast transmission, Missive should add minimal additional time moving an app from the trusted cache into an executable condition in a new process; this is its buffer-cache-like function.

Figure 14 shows time to launch Gimp and several Midori-based apps, for which we have good internal probe points and can measure startup time all the way to the point of user interactivity. It contrasts launching inside Embassies with starting the same content on Linux. The "hot" start represents the primary function of a buffer cache: bringing a machine-resident app into memory for prompt execution. Missive's hot start is about 100 ms or less overhead.

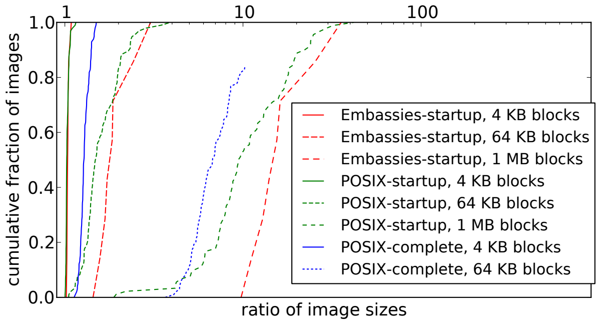


Figure 13: **Small-file packing is necessary.** *Giving small files their own blocks maximizes sharing opportunity, but the file distribution includes so many small files that doing so generates significantly more padding than the actual file content. The smallest block sizes have median bloat of 2–28%.*

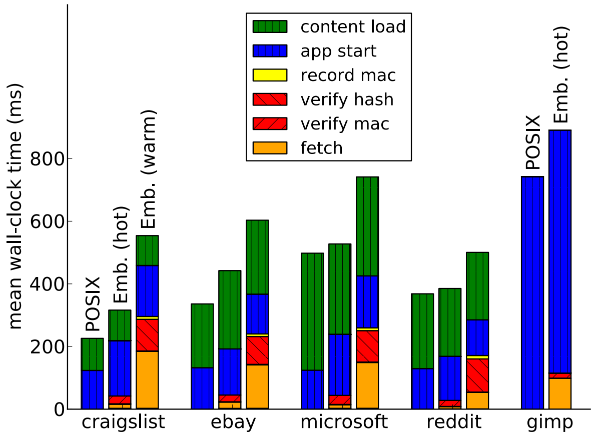


Figure 14: **Missive's launch is comparable to POSIX buffer cache.** *In the hot case, the app's blocks are in the untrusted cache; the primary overhead is the MAC verification (§3.3.2). In the warm case, missing app blocks are fetched from a fast server; the primary overheads are the slower hash and the memcpys required to assemble the image (§3.1.2).*

A 100 ms delay may seem expensive [26], but three factors mitigate it. First, it only affects the launch of a new binary; navigation among pages or activities within a site's application are unaffected. Second, once a site's binary is running, the vendor controls both ends; it is free to deploy SPDY [30] or the next innovation anytime. Third, we have only performed black-box optimizations; application-specific tuning could dramatically reduce the amount of data needed for startup to the first point of interactivity.

In the "warm" start case, the cache contains a copy of the Midori browser with a vulnerable version of libpng. We measure the time to start an app using a similar stack (exploiting that local commonality) with a patched libpng. Since the network overhead is a function of content size (studied in Figures 7 and 8), this experiment uses a local network connection to focus attention on the

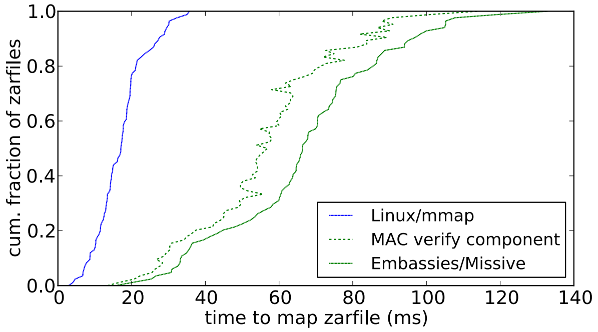


Figure 15: **Contrasting just launch time** *across the broader Posix-initial data set, the median Missive app takes about 50 ms longer, most of which is verification time. (Verification times are sorted with the corresponding app, not cumulatively.)*

system's inherent sources of latency. In the warm start case, Missive's cache requires about 150 ms to assemble the outgoing zarfile from cached blocks, and the receiver pays as much again to verify the zarfile with a hash.

Figure 15 measures the broader Posix-startup data set, but more shallowly, in that it only captures the cost of mapping the executables into memory. For this hot start experiment, we had a Linux process `mmap` every file in the data set and `read` the rest; we contrasted that to Missive's launch step, which pulls one zarfile into its memory. In both cases, the test apparatus touched every memory page. The median Linux time is 17 ms. Most of Missive's 66 ms is the cost of integrity verification. Although the cost is 4× higher in relative terms, the overall burden is not overwhelming compared to the overall start time of typical applications, which this experiment excludes.

While Missive is not completely without cost, in the hot case where it can use VMAC (§3.3.2), it hovers very close to the performance of a native, trusted buffer cache.

6 Related Work

Numerous systems have proposed content-addressable techniques for identifying and routing content, as well as reducing bandwidth [5, 7, 11, 14, 28, 39]. We briefly touch on some of the most related efforts below.

Tolia et al. proposed a content-oriented data transfer service with the aim of decoupling application-level content negotiation from data transfer, hence enabling greater innovation in transfer protocols [38]. Like Missive, they divide objects into chunks identified by hash, allowing caches to identify shared content across applications. The receiver drives the data transfers by specifying chunks of interest. Since Tolia et al. focus on issues related to data transfer, they do not consider, as Missive does, how to identify and package app-related files, they assume the local content cache is trusted, and they do not address the final step of rapidly transferring the app image into a booting process and verifying it.

Van Jacobson et al. and Trossen et al. have proposed building efficient commonality-exploiting data transfer by addressing content at the network layer [19, 40]. Rhea et al. use content hashes to reduce the bandwidth needed for Web traffic [34]. Spring and Wetherall also use hash-based matching to perform data deduplication at the IP layer [35].

Tangwongsan et al. propose a multi-resolution handprint for selecting a content chunk size that optimally exploits data redundancy in files [37].

Multiple projects (e.g., the Collective [6] and Internet Suspend/Resume [22]) proposed distributing applications as full VMs, both to simplify management and to minimize cross-application conflicts. On the server side, SnowFlock [24] proposed a data-center-wide VM fork operation for instantiating a single VM on hundreds of machines. To minimize launch latency, they developed several techniques for lazily replicating only the active working set of the VM being forked. Unlike Missive, these projects assume a trusted local cache.

The deduplicating transport of Missive is constructed of fairly conventional techniques. We considered using BitTorrent [8] or other deduplicating transports [13, 28]. Using a custom transport protocol, however, admits three advantages: its use of Merkle trees enables immediate use of partially-loaded images, it is optimized around the untrusted cache's extreme latency constraints, and its simplicity enables the same protocol to function well in the wide-area and be implemented in a tiny boot block.

The most crucial abstract idea in Missive is the separation of sharing content for performance from sharing by reference; this distinction was significantly inspired by the Slinky system [9].

7 Conclusion

By virtue of the protocol that boot blocks use to share it, Missive's untrusted cache supplants functions normally supplied by a host's (very trusted) buffer cache. It coordinates memory as a cache for disk, and disk as a cache for the network. It rapidly assembles an application's image from parts both unique and common with other apps. It exposes sharing in a way that the underlying memory page manager can exploit for performance, while preserving for apps the abstraction of private copies; but because naming is by content, the isolation is much stronger than in the conventional use of a buffer cache.

References

[1] ADAMS, S. Smaller is faster (and safer too). `http://blog.chromium.org/2009/07/smaller-is-faster-and-safer-too.html`, July 2009.

[2] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R.,

AND LORCH, J. R. A five-year study of file-system metadata. *Trans. Storage 3*, 3 (Oct. 2007).

[3] BAKKER, A. Merkle hash torrent extension. http://www.bittorrent.org/beps/bep_0030.html, Mar. 2009.

[4] BAUMANN, A., LEE, D., FONSECA, P., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing OS extensions safely and efficiently with Bascule. In *EuroSys (to appear)* (2013).

[5] BRESSOUD, T. C., KOZUCH, M., HELFRICH, C., AND SATYANARAYANAN, M. OpenCAS: A flexible architecture for content addressable storage. In *Workshop on Scalable File Systems and Storage Technologies* (Sept. 2004).

[6] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C. P., AND LAM, M. S. The Collective: A cache-based system management architecture. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (May 2005).

[7] COHEN, B. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems* (June 2003).

[8] COHEN, B. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer systems* (2003), vol. 6, pp. 68–72.

[9] COLLBERG, C., HARTMAN, J. H., BABU, S., AND UDUPA, S. K. Slinky: static linking reloaded. In *USENIX ATC* (2005).

[10] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for Web applications. In *IEEE Symp. on Security & Privacy* (2006).

[11] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2001).

[12] DOUCEUR, J. R., AND BOLOSKY, W. J. A large-scale study of file-system contents. In *SIGMETRICS* (1999).

[13] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. The Utility Coprocessor: Massively parallel computation from the coffee shop. In *USENIX ATC* (2010).

[14] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peerto-peer storage utility. In *Proceedings of the HotOS Workshop* (May 2001).

[15] FARSHAD. Best 60 linux applications for year 2011. http://www.addictivetips.com/ubuntu-linux-tips/best-60-linux-applications-for-year-2011-editors-pick/, Jan. 2012.

[16] FU, K., KAASHOEK, M. F., AND MAZIÈRES, D. Fast and secure distributed read-only file system. In *OSDI* (2000).

[17] HOWELL, J., DOUCEUR, J. R., ELSON, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008).

[18] HOWELL, J., PARNO, B., AND DOUCEUR, J. Embassies: Radically refactoring the web. In *NSDI (to appear)* (2013).

[19] JACOBSON, V., SMETTERS, D. K., THORNTON, J. D., PLASS, M. F., BRIGGS, N. H., AND BRAYNARD, R. L. Networking named content. In *5th international conference on Emerging networking experiments and technologies (CoNEXT)* (2009).

[20] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., NO, H. M. B., HUNT, R., MAZIÈRES, D., PINCKNEY, T., GRIMM, R., JANNOTTI, J., AND MACKENZIE, K. Application performance and flexibility on Exokernel systems. In *SOSP* (1997).

[21] KATABI, D., HANDLEY, M., AND ROHRS, C. Congestion control for high bandwidth-delay product networks. In *SIGCOMM* (2002).

[22] KOZUCH, M., AND SATYANARAYANAN, M. Internet suspend/resume. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications* (June 2002).

[23] KROVETZ, T., AND DAI, W. VMAC: Message authentication code using universal hashing. Internet Draft: http://http://fastcrypto.org/vmac/draft-krovetz-vmac-01.txt, Apr. 2007.

[24] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. SnowFlock: Rapid virtual machine cloning for cloud computing. In *EuroSys* (Apr. 2013).

[25] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. TrInc: Small trusted hardware for large distributed systems. In *NSDI* (2009).

[26] LINDEN, G. Make data useful, 2006. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf.

[27] MERKLE, R. C. A certified digital signature. In *CRYPTO* (1989), pp. 218–238.

[28] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP* (2001).

[29] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *SOSP* (2005).

[30] PADHYE, J., AND NIELSEN, H. F. A comparison of SPDY and HTTP performance. Tech. Rep. MSR-TR-2012-102, Microsoft Research, July 2012.

[31] PARNO, B., LORCH, J. R., DOUCEUR, J. R., MICKENS, J., AND MCCUNE, J. M. Memoir: Practical state continuity for protected modules. In *Symposium on Security and Privacy* (2011).

[32] PARNO, B., MCCUNE, J. M., WENDLANDT, D., ANDERSEN, D. G., AND PERRIG, A. CLAMP: Practical prevention of large-scale data leaks. In *Symposium on Security and Privacy* (2009).

[33] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the Library OS from the Top Down. In *ASPLOS* (2011).

[34] RHEA, S. C., LIANG, K., AND BREWER., E. Value-based web caching. In *Proceedings of the World Wide Web Conference* (May 2003).

[35] SPRING, N. T., AND WETHERALL, D. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM* (Sept. 2000).

[36] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. File size distribution on UNIX systems: then and now. *SIGOPS Oper. Syst. Rev. 40*, 1 (Jan. 2006), 100–104.

[37] TANGWONGSAN, K., PUCHA, H., ANDERSEN, D. G., AND KAMINSKY, M. Efficient similarity estimation for systems exploiting data redundancy. In *Proceedings of IEEE INFOCOM* (Mar. 2010).

[38] TOLIA, N., KAMINSKY, M., ANDERSEN, D. G., AND PATIL, S. An architecture for internet data transfer. In *Proceedings of USENIX NSDI* (May 2006).

[39] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., PERRIG, A., AND BRESSOUD, T. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2003).

[40] TROSSEN, D., SARELA, M., AND SOLLINS, K. Arguments for an information-centric internetworking architecture. *SIGCOMM Comput. Commun. Rev. 40*, 2 (Apr. 2010), 26–33.

[41] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity and containment in the Potemkin virtual honeyfarm. In *SOSP* (2005).

[42] WHEATLEY, R. Top 100 of the best (useful) opensource applications. http://www.ubuntulinuxhelp.com/top-100-of-the-best-useful-opensource-applications/, Feb. 2008.

[43] 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 *Symposium on Security & Privacy* (2009).

A Modular and Efficient Past State System for Berkeley DB *

Ross Shaull
NuoDB

Liuba Shrira
Brandeis University

Barbara Liskov
MIT/CSAIL

Abstract

Applications often need to analyze past states to predict trends and support audits. Adding efficient and non-disruptive support for consistent past-state analysis requires after-the-fact modification of the data store, a significant challenge for today's systems. This paper describes Retro, a new system for supporting consistent past state analysis in Berkeley DB. The key novelty of Retro is an efficient yet simple and robust implementation method, imposing 4% worst-case overhead. Unlike prior approaches, Retro protocols, backed by a formal specification, extend standard transaction protocols in a modular way, requiring minimal data store modification (about 250 lines of BDB code).

1 Introduction

Applications need *retrospection*, the ability to analyze past states, to provide audits and predict trends. Without adequate support in the data store, it is hard for developers to reconstruct consistent past states corresponding to events of interest. Yet, many data stores lack support for retrospection because adding a low-impact consistent past state system to a data store has been challenging using current approaches.

This paper describes Retro, a system that adds retrospection to Berkeley DB (BDB), a popular transactional key-value database. Our system automatically stores past states of interest to the application, and allows the applications to query automatically restored consistent past states. The queries can take advantage of the application code base; any read-only application or library program that runs in BDB can also run retrospectively.

The novel contribution of Retro is an efficient yet simple and robust implementation method. Retro is very efficient; it does not disrupt BDB performance even when applications retain the past states at high frequency, imposing a minimal 4% performance penalty in the worst case. Furthermore, Retro extends standard database protocols in a modular and robust way, based on a simple past state system specification that serves as a basis for a formal proof of Retro protocol correctness (presented in [13]); this is in contrast to prior past state systems (e.g., [5, 15, 17]), which used more complex and fragile ad-hoc modifications to data store internals to avoid disrupting database performance. The code implementing Retro protocols composes with standard interfaces in the database software stack. Because the composition is modular, the modifications to the database code are minimal: our prototype modifies only 250 lines of Berkeley DB source code.

To preserve transaction performance when saving consistent past states, Retro captures the needed past states incrementally, using split copy-on-write [15]. It then creates the past state lookup structures [14] and accumulates the past states and lookup structures in additional memory that is dedicated to storing past state information. Retro writes the past states and lookup structures to a separate log-structured store, in the background without interfering with database queries. Retro recovery ensures the past states and lookup structures remain consistent and durable in the presence of crashes. To query past states, Retro redirects code to snapshot pages using dynamic translation structures, without interfering with database transactions.

Retro accomplishes its tasks by extending BDB update commit, update recovery, and update writing and reading protocols in a simple and intuitive way. An extension to the commit protocol retains the needed past states when an update transaction commits. Similarly, an extension to the BDB recovery protocol retains the needed past states when BDB recovers updates from the transaction log after a crash, thus relegating past state recovery to database recovery, an important simplification since recovery pro-

[1]This work was partially supported by the National Science Foundation under grants NSF IIS-1251037, NSF CNS-1318798. The work was accomplished when Ross Shaull was at Brandeis University.

tocols can be complex.

Retro recovery faces two complications. Care must be taken to ensure that BDB does not discard the transaction log before past states become durable, and to ensure that recovery attempts that fail and restart do not corrupt past states. These dangers are avoided by enforcing a simple invariant ensuring that past states become durable before database updates.

Retro is implemented as a system of concurrent callbacks running as part of BDB commit, recovery, and writing and reading protocols. The concurrent callbacks access shared state, e.g., to track which past states have been already saved. The callbacks must therefore run in a thread safe manner, and moreover must be serialized in a consistent order to ensure consistent past states can be identified correctly. If Retro callbacks were to block unnecessarily, this could increase BDB transaction latency. To avoid this bottleneck, Retro stores its shared state in specialized data structures that eliminate blocking.

In summary, the contributions of this paper include 1) a new efficient system for retrospection in Berkeley DB, implemented using a simple and robust method, avoiding invasive database modifications. 2) modular past state protocols justified by a formal specification that extend standard transaction protocols, 3) design of the modular snapshot layer that implements the past state protocols, using specialized data structures that minimize overheads to BDB, 4) experimental evaluation supporting our efficiency claims, and analysis of retrospection performance for in-memory and on-disk past states.

Retro was designed for BDB but we believe our method is general and can be applied to other transactional data stores, contributing a step towards making past state support more widely available to applications.

2 Related Work

Past states can be supported at the level of logical records or files (e.g., [5, 12]), or low-level pages (e.g. [17]), like Retro. Without close integration with the data store, past state systems can impose a high performance penalty to provide transactionally consistent records, or crash consistent files. Systems that operate *below* the data store (e.g., page-level Windows VSS), block update transactions, disrupting performance if snapshots are frequent. Temporal databases that operate *above* a database, restrict scalability [9].

Integrated past state systems can exploit data store mechanisms to write consistent past states at low cost. Postgres [3] and versioning file systems (e.g., [4]) integrate with a no-overwrite system that keeps past records in place, and copies new records to a new place, reducing the number of writes. Since past and current state are not separated, large past state can negatively impact the cost of current state reads [3]. Read impact can be avoided by keeping the past state separate. Ganymed [10], a Postgres based system, copies past records to a separate Postgres replica node, using replication middleware. The replica provide access to historical snapshots using modified concurrency control and query protocols. Read impact can also be avoided by exploiting recovery (e.g. fuzzy checkpoints [3]), but recovery based methods are too slow for on-line programs.

ImmortalDB [5] supports consistent past records, integrating with SQL Server. The database data layout is modified to keep recent versions of past records on the current state pages, eventually migrating old versions to separate pages; indexes are modified to support temporal access. Oracle Total Recall supports integrated historical record tables, indexed like regular tables. SNAP [15] supports consistent indexed split page-level snapshots, integrating with an object store. SFS [17], supports split page-level snapshots, integrating with a file system. All above systems integrate past state support using invasive modifications to the data store internals. VersionFS [7] adds versioning in a stackable file system. The stackable architecture supports modular extension, a goal shared by Retro .

Retro adopts the split snapshot representation in SNAP [15] and Skippy index [14], extending the prior work in important ways. The Skippy work provides efficient multi-level index for split snapshots, without considering index recovery, concurrency, or implementation of a complete snapshot system. Retro implements a complete snapshot system, including efficient recovery and non-blocking concurrency control protocols for Skippy index and snapshot data. Unlike SNAP, and other implementations using invasive ad-hoc modifications, Retro provides an efficient implementation method based on modular extension of standard transaction protocols. The modular snapshot protocols, based on a formal snapshot system model, and their low-cost implementation structures in BDB are the new contributions of Retro.

3 Programming Model

From the application developer's perspective, programming with Retro is a straightforward extension to programming with BDB using a simple *named* snapshot abstraction (Figure 1).

Retro supports C and SQL (SQLite) BDB APIs. Applications run transactions that issue update and query requests to records organized in tables. An applications may declare a persistent snapshot at transaction boundary at any time by issuing the *snapshot now* command. The declaration command commits a transaction, whose serialization point defines the contents of the snapshot. A snapshot represents the state of the entire database (e.g.,

Current-state queries are unchanged by Retro
results ← select * from ...
Applications may declare snapshots at any time and get back a snapshot identifier
S ← snapshot now
As of queries are delimited with snapshot identifier
results ← as of **snapshot S** { select * from ... }

Figure 1: Programming with snapshots

tables, indexes, system catalogs).

After declaration commits, the application is returned a *snapshot identifier* which permanently identifies the declared snapshot. A snapshot identifier may be used to access a snapshot immediately; no delay is required between declaring and accessing a snapshot. Retro does not mandate how snapshot identifiers are remembered for later use, e.g. they can be stored in a table along with a timestamp. Internally, Retro assigns to snapshot identifiers consecutive integer names in declaration commit order.

To make use of the snapshots they declare, applications run queries as of those snapshots. The application delimits any *read-only* query code with *as of* to instruct Retro to run the delimited query retrospectively. A query delimited by *as of snapshot S* reflects the effects of all the transactions committed prior to the serialization point of *S*, and none of the effects of transactions committed after *S*. Inside the *as of* delimiter, the query itself is written just like a normal, current-state query. This makes it easy to leverage existing programmer knowledge and codebases when programming with Retro.

Existing current-state BDB code runs unmodified. Code that declares snapshots and runs retrospection can be executed alongside the current-state code in the same database, making it easy to use retrospection from current state BDB code on-line where needed.

4 Retro architecture

The relevant components of BDB software stack are depicted in Figure 2. The shaded areas show Retro extensions (explained later). A BDB application runs transactions that manipulate logical data records and tables, issuing update and query requests against the BDB API, which processes the requests and translates them into requests to the transactional *storage manager*. The storage manager manages logical data records organized in pages, stored on durable storage. The pages are the unit of transfer between durable storage (on disk) and the page cache (in memory).

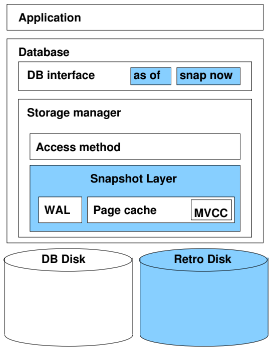


Figure 2: Retro modularized in database architecture

When an application requests a data record as part of some query (e.g., “get record k”), that request is processed by an *access method* (a binary tree or a hash table), a storage manger component that encapsulates the representation of records in a table. The access method translates requests to read and write data records into requests to the page cache to read and write data in units of pages. Pages are cached in the page cache in memory, and data records are read from and written to the cached pages.

A request for a page issued to the page cache by an access method refers to the page by its *logical name*. A logical name is a pair $(file, number)$, where file is the identifier of a database file open in the page cache, and number is some offset within that file. The logical page name gets translated to its physical disk address when page cache performs disk I/O.

The storage manager includes two additional component relevant to Retro, concurrency control and recovery, responsible for transaction serialization and crash consistency, respectively. These components will be described later when we explain how Retro extends them.

4.1 BDB components extended by Retro

The mechanisms that implement Retro functionality are modularized within the *snapshot layer* (SL) (figure 2). The SL wraps components in the transactional storage manager, extending BDB behavior to add efficient snapshot creation and querying. Snapshots are created and accessed by extending the page cache and concurrency control (section 4.4 and section 5). Snapshot recovery supporting efficient snapshot creation is achieved by extending recovery (section 6).

Retro does not affect access methods, the logical-to-physical translation of page names into their disk addresses, or how the database is organized on disk. These storage manger components are transparent to Retro. Retro is concerned with logical page names, and expanding that namespace to support snapshots.

Retro assumes that page cache memory and extra storage is allocated for holding snapshots. The extra page cache memory allows the system to accumulate snapshots before writing them to disk without contending with current state transactions. For best performance, the extra storage is on separate disks from the database, to avoid interference. Snapshots have the potential to consume much more storage that the current state, since (generally speaking) history grows with any insert, update, or delete, while the current state only grows when data is inserted. Retro helps use an investment in additional storage efficiently by being selective (Retro keeps only declared snapshots) and incremental (snapshots may share snapshot pages). Currently, Retro does not allow snapshot deletion. Adopting low-cost snapshot deletion [16] is a straightforward extension.

4.2 Low-level snapshots

At the level of the storage manager, a snapshot S is a *complete* and *transactionally consistent* collection of *snapshot pages*. The collection is complete because it contains all the data and metadata pages in BDB, including index and catalog pages. The collection is transactionally consistent because the snapshot pages reflect the serialization point of the snapshot declaration command, making snapshot pages consistent with one another. These properties allow any new or existing read-only code that could run in the database when snapshot S was declared, to run *as of* a snapshot S.

The page-level Retro snapshot abstraction makes it possible to run retrospective queries without changing access methods, indices, and other storage system code that relies on page layout and naming. The same name used to denote a page in the current state is used during retrospection to denote a snapshot version of that page. This means that snapshot pages that refer to each other by name (e.g., index pages) can be used normally by storage system code.

For convenience, we denote the version of a page P as of a snapshot S using $P@S$. This is purely notational; when BDB requests a page P while querying restrospectively as of S, it requests P using the same page name as it would if it were requesting P as part of a current-state query. Section 4.4 describes this mechanism in detail.

4.3 Snapshot overwrite sequence

Retro creates snapshots using split copy-on-write [15] and Skippy index [14]. When a snapshot is declared, all snapshot pages are *shared* with the current database pages. When a BDB transaction updates a page that is *shared* with a snapshot, Retro saves the snapshot page in memory, updates the snapshot indexing metadata, and eventually writes pages and metadata to the Retro disk.

We use the notion of *Snapshot Overwrite Sequence* (OWS) to reason about what Retro protocols need to do, i.e. as a correctness specification.

> **Definition**: Let H be a serial committed transaction execution history. The snapshot *overwrite sequence* of H (OWS(H)) is a mark up of H that tags every snapshot declaration and every commit that updates a page *shared* with a snapshot.

OWS(H) captures the points in the execution sequence where snapshot pages and index metadata must be created. There is a straightforward formal proof of correctness of Retro protocols based on OWS [13], omitted for lack of space. OWS answers the following questions: **1) which page versions to save**: the page pre-state of the first update to a page following a snapshot declaration (Sec 5), **2) what state to recover after a crash**: a recovery after a crash immediately following execution H must recover all pages and metadata created by operations tagged in OWS(H) (Sec 6), **3) where to get the page P@S requested by a retrospective query running after H**: from Retro, if there is a tagged commit updating P following the declaration of S in H, or otherwise from the database (Sec 7).

For example, consider the following OWS(H):

$$T1(S1)T2(wR)T3(S2)T4(wRwQ)$$
$$T5(wRwQ)T6(S3)T7(wRwP)$$

Retro saves pre-states in $T2$ (R), $T4$ (R,Q), and $T7$ (R,P) but not $T5$ since its updates to R and Q are not the first modifications to a page following the declaration of snapshot $S2$. For a crash following H, Retro recovers all the pre-states and indexing state created in $T1$ to $T4$, $T6$ and $T7$. A retrospective query as off snapshot 3, when issued after H, gets $P@3$ from Retro, but when issued before $T7$, gets $P@3$ from the database.

4.4 Logical page virtualization

Retro allows retrospective queries to execute concurrently with current state queries and updates in the same page cache, allowing current state programs to directly query past states. The key idea behind the execution architecture for retrospection is to translate the BDB logical page names to the names of snapshot pages when a query runs retrospectively, using logical page virtualization in the snapshot layer (SL). The name translation is transparent to BDB, enabling storage system code using logical page names to run on both the current state and snapshots.

Figure 3 depicts the retrospective query execution path. Retro stores the snapshot pages in a file called

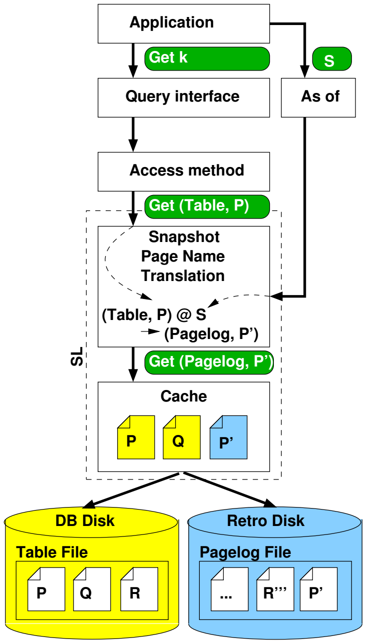


Figure 3: Retrospective query execution

Pagelog located on the Retro disk. This file is opened in the page cache like any other database file. A snapshot page $P@S$ that has been overwritten since S was declared will be copied to Pagelog. For brevity, we refer to the offset in Pagelog where $P@S$ is stored as P'; the actual offset P' has no relationship to the logical name of P in the current state.

The logical page virtualization is implemented by a *translation component*. The translation component intercepts page requests from access methods and translates logical database page names into logical snapshot page names if the code issuing the request is running retrospectively. The *as of* primitive provided by the Retro interface extension identifies the snapshot from which a requested page should be read.

The translation component keeps track of translations from logical names (e.g., page P in a database file *Table*) to pre-states in Pagelog (e.g., P' in *Pagelog*) for any declared snapshot. After translating $(Table, P)@S$ to $(Pagelog, P')$, the translated name is passed to the page cache, which reads and caches it like any other page. The contents of the snapshot page $(Pagelog, P')$ are returned to the access method as though it were the current-state contents of the page named $(Table, P)$, with the access method and application none the wiser that the requested page was transparently switched with a snapshot page. If the retrospectively accessed page named $(Table, P)$ is still shared with the database, the name translation in SL will be identity function, and the logical name $(Table, P)$ will be requested from the cache.

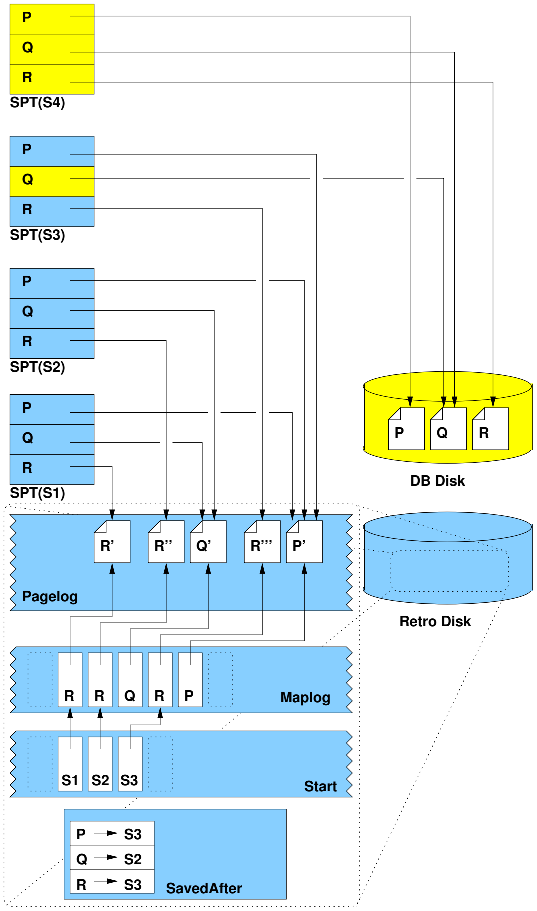


Figure 4: Snapshot representation

BDB programs that read and write the current state are unaffected by Retro. When the application performs a query without specifying *as of*, Retro does not engage page name translation, and the logical page name is passed unchanged to the page cache. Because different versions of the same page have distinct logical names, they can coexist in the cache.

4.5 Retro snapshot representation

Retro adapts the *split snapshot* representation [15, 16] to organize snapshot data (pre-states) and metadata (indices for page name translation) . Split snapshots are created at the page level, and stored separately ("split") from the database (e.g., on a separate disk). By splitting snapshots from the database, the approach does not affect how the database organizes data, and partially isolates database I/O performance from snapshot-related I/O.

Figure 4 depicts the organization of Retro data and metadata. When an update U tagged in the OWS commits a modification to a page P, Retro saves in memory the the *pre-state* of P created by U. For every saved pre-state, Retro also updates snapshot metadata in memory to note the snapshots to which the pre-state of P belongs. Snapshot metadata includes *Maplog*, which maps from logical page names *as of* a declared snapshot to the location of pre-states in Pagelog; *SavedAfter*, which tells if a snapshot shares a particular page with the current state by tracking for every database page the latest snapshot for which a pre-state of that page was saved; and *Start*, which associates every declared snapshot with the first Maplog entry created for that snapshot.

Snapshot data and metadata are stored on disk. The on-disk representation in Figure 4 corresponds to the example transaction history in Sec 4.3. Section 6 explains how Retro writes snapshot data and metadata to disk in efficient and recoverable manner.

Snapshot Page Tables Snapshot page tables (SPTs) are in-memory tables used to implement snapshot page translation during retrospection (section 4.4). SPT entries map the logical page names in the database to the logical names of snapshot pages that can be either pre-states in Pagelog, or pages shared with the database. Resolving the snapshot page name $P@S$ to the logical name of a page in Pagelog requires looking up P in SPT(S). In Figure 4, SPT($S1$) and SPT($S2$) share Q' since there was no update to Q between S1 and S2. In SPT($S3$), Q points to the database because Q has yet been modified since $S3$ was declared.

Maplog, Start, and SavedAfter Keeping SPTs in memory for every declared snapshot would be costly, so instead Retro reconstructs SPTs from the saved metadata. When a retrospective query is run as of S, Retro builds SPT(S) by scanning Maplog for the first occurrence of a mapping for each page; these first-encountered mappings correspond to the pre-states saved from update tagged in the OWS. Retro scans Maplog from Start (S) since earlier mappings correspond to pre-states saved before S was declared. A scan will not encounter mappings for snapshot pages that are still shared with the database. Retro can determine this, without scanning to the end of Maplog, from the data structure SavedAfter.

E.g., in figure 4, SavedAfter shows pages P and R were last saved after $S3$ was declared, but the snapshot after which Q was last saved is $S2$.

Naively scanning Maplog can be expensive, because mappings for pages for which pre-states are frequently saved (due to update skew) increase the length of a Maplog scan. An efficient indexing technique to combat the impact of update skew called Skippy, is described in [14].

An SPT built for a retrospective query Q running as of snapshot S needs to be kept up to date as Retro saves snapshot pages. The techniques used to accomplish this are discussed in [13].

5 Extending MVCC

BDB serializes transactions using a popular multi-version page-level concurrency control protocol (MVCC) that enables concurrent transaction reads and updates by keeping multiple versions of pages in memory. Every transaction T "sees" a consistent view of the database, called an isolation snapshot, consisting of the versions of pages which were most-recently committed *before* T began. If T updates a page, the new page version becomes visible to other transaction after T commits. MVCC versions pages in the page cache [1, 8] in the storage manager.

Retro extends MVCC, creating persistent snapshots out of isolation snapshots. To avoid confusion, we refer to the persistent snapshots created by Retro as *Psnaps* and the volatile isolation snapshots created by MVCC as *Vsnaps*.

MVCC mechanisms When an application requests a page P for *update* (via an access method) as part of transaction T, a private copy of P is created in the cache and associated with T. When T commits, P is marked with a number called the *commit LSN* of T (Sec 6), identifying the last transaction that updated P. The versions of P are linked together in a list called the *version chain*.

When an application requests to *read* a page P (via an access method) as part of transaction T, the page cache searches the version chain for the latest version committed before T began. If the page cache has no versions of a page P, P is read from the disk. To avoid overflowing the cache, MVCC garbage collects page versions on each version chain that are no longer visible to any running transactions. Every transaction has an associated Vsnap, and Vsnaps of multiple transactions may coexist in the same cache.

5.1 Persisting snapshots

To persist a Psnap S, Retro needs to save the latest version of every page committed before S was declared, and eventually write it to Pagelog on the Retro disk.

Retro saves pages from S using page-level copy-on-write in the page cache. It relies on MVCC to create, in memory, the needed page copies in Vsnaps of transactions that update pages following a snapshot declaration, and uses SavedAfter data structure to identify the needed versions among these copies. SavedAfter relates every logical page name to the latest snapshot after which a version of it was last saved. When a transaction T commits, Retro checks SavedAfter to see if any page P updated by T is the first update to be committed since the latest-declared snapshot S, and if so, saves the pre-state of P created by T and updates SavedAfter.

Retro writes the saved pre-states efficiently using asynchronous I/O. However, pre-states which are no longer visible in any Vsnap are candidates for MVCC garbage collection. So, Retro must maintain the following invariant to ensure that pre-states needed for a declared Psnap can be written to Pagelog:

> **Before-GC invariant**: any pre-state from an update in the OWS is saved for Pagelog before MVCC garbage collects it.

Retro maintains this invariant by writing the pre-states in the Vsnap of a transaction T lazily after T commits, postponing garbage collection of the pre-states for a short time if necessary. Since MVCC garbage collection also happens lazily when page replacement is needed, the delay has minimal impact.

6 Extending Recovery

Snapshot recovery is greatly simplified by leveraging the native BDB recovery mechanism. When BDB recovery replays the updates of transactions that committed before the crash, Retro saves the intermittent page versions that correspond to the pre-states of updates tagged in OWS, mimicking snapshot creation during normal execution. This way, if BDB creates intermittent page versions identical to the ones created during normal execution, when recovering a history H, Retro will create snapshot pages and metadata defined by OWS(H).

During recovery, BDB produces the same intermediate versions that were produced by the earlier execution as long as the page versions read from disk after the crash are the same versions used by original updates. Care must be taken to ensure that snapshots are correctly recovered. BDB may discard the update records from the log after updates are written to the database. Shapshots could be lost if updates records are discarded prematurely. Moreover, if BDB recovery crashes while updating the database pages on disk, then when recovery restarts, it may encounter disk page versions different from those needed by OWS, causing snapshot pages to be lost.

Retro avoids the complications by enforcing the following simple write-ordering invariant, called the *write-ahead snapshot invariant*, during normal operation and during recovery:

> **Write-ahead snapshot (WAS) invariant**: The pre-state of P (and associated snapshot metadata) from an update in OWS(H) must be written to the Retro disk before the version of P created by that or any later update in H is written to the database disk.

WAS invariant guarantees snapshot pages and metadata become durable before database pages needed to recover them become overwritten.

During replay (like in normal execution), Retro consults snapshot metadata e.g., SavedAfter, to check if the pre-state needs to be saved. Snapshot metadata therefore, must be recovered the database. Retro simplifies metadata recovery by storing all metadata (Start, Maplog, and SavedAfter) in BDB transactional data structures, and updating metadata using regular transactions, Retro relies on BDB to recover metadata.

It would be costly to write snapshot data using a database transaction, since it is large compared to snapshot metadata. Instead snapshot data is written using regular writes. So, the Retro recovery protocol enforces a second write-ordering invariant to make it possible to clean up partially-written snapshot data after a crash and correctly detect whether a particular pre-state was written to the Retro disk before the crash:

> **Snapshot-data-before-metadata invariant**: Before Retro commits the mapping for pre-state P', it writes P' to Pagelog.

The Retro write invariants, enforced during normal operation and recovery, and the two stages of Retro recovery (metadata and replay), guarantee that for every BDB recovery of transaction history H, Retro will correctly recover snapshot data and metadata defined by OWS(H), even in the presence of repeated BDB recoveries.

6.1 BDB recovery

BDB follows the write-ahead logging (WAL) protocol to ensure recoverability after a crash. The WAL protocol requires that the database write a record of updates made by a transaction T in a durable log before T commits. Each update is represented by a log record which contains (at least) the information required to repeat the update; this is called a REDO record. We assume that log records are not coalesced across transactions; i.e., every (committed) page update has a corresponding REDO record. Log records are ordered and identified by a monotonically-increasing log sequence number (LSN). Transaction commit is recorded in the log using a commit record. The transaction commit record LSN determines the transaction serialization order.

Periodically, the database performs *checkpoints*. A checkpoint C writes page versions committed prior to some chosen checkpoint LSN (LSN_C) to the database disk, and then records the checkpoint LSN in the log. To simplify the description, we assume the database only performs writes during a checkpoint. Retro, however, supports general database write policies (i.e. writes due to cache pressure) not described here, for lack of space.

We assume the database follows a no-STEAL writing policy [6]. This means that the database never writes uncommitted updates to disk. The recovery, therefore, only needs to REDO updates that were committed but not yet written since the last checkpoint; a database with a STEAL writing policy must also *undo* changes to pages which were not yet committed before the crash. Large memories render STEAL policy less important.

Checkpoints bound recovery time; the database does not need to recover updates that have been written to the database disk. The database is free to garbage collect the log entries prior to LSN_C, and begins recovery from LSN_C (to be more precise, the database starts recovery at, and can garbage collect log entries older than, the LSN of the oldest transaction that began before LSN_C and committed after LSN_C).

After a crash, BDB enters *recovery* upon being restarted. The database cannot re-enter normal operation until recovery has completed. During recovery, the database *replays* committed updates by applying REDO records from the WAL in LSN order since the last checkpoint. A successful recovery ends with a checkpoint; after recovery, the on-disk state of the database reflects the history of transactions that committed prior to the crash.

6.2 Snapshot recovery details

Retro expects to be invoked when BDB recovery applies REDO records. To identify updates tagged in OWS Retro must order updates relative to snapshot declarations. Retro records snapshot declarations in the log during normal execution, using a special log record (a *snaprec*). Retro expects to be invoked when recovery encounters snaprecs so that Retro can handle them (i.e. re-declaring the snapshot if needed).

Because BDB uses page-level concurrency control (i.e., two overlapping transactions may not both commit an update to the same page), we know that REDO records for a page *P* will be applied in transaction commit order. Retro invocations from update replay and snapshot declarations therefore run in transaction commit order, making it easy to identify updates tagged in OWS.

Algorithm 1 Retro extension to database checkpoint

```
Pause database checkpoint
Write unwritten snapshot data to disk
Transactionally update snapshot metadata created
    since the last database checkpoint
Allow database checkpoint to proceed normally
```

Algorithm 1 shows how the write-ordering invariants are enforced during a checkpoint. Whenever the database initiates a checkpoint, Retro takes control and finishes writing any snapshot data that had not yet been trickled to disk, and then updates snapshot metadata atomically using a database transaction. Finally, Retro returns control to the database, allowing the checkpoint to proceed normally.

Retro allows snapshot data to be trickled during normal operating periods (snapshot data is large and grows in proportion to the number of updates, so it is impractical to buffer all snapshot data created between checkpoints). After a crash, some pre-states may be on the Retro disk that have no corresponding snapshot metadata but the reverse can never be true due to snapshot-data-before-metadata write invariant. This means that when Retro recovery begins, any snapshot data written since the last checkpoint can be deleted by deleting any pre-states that are not referenced from snapshot metadata.

Algorithm 2 shows the two stages of Retro recovery. Stage 1 of Retro recovery resets snapshot data and metadata to a consistent state; then, Stage 2 runs alongside BDB recovery, saving a needed pre-state (if it has not been saved already). Database recovery ends with a checkpoint, the completion of which marks the completion of a successful recovery. Retro enforces the write invariants WAS and snapshot-metadata-before-data during this checkpoint, just like during normal operation. So, the checkpoint that terminates a successful database recovery for history H also marks the end of Retro recovery, at which point the on-disk state of Retro will reflect OWS(H).

Retro needs to *suppress* duplicate re-creation of snapshots in repeated recovery. Retro will know to correctly suppress re-creation of snapshots because after Stage 1, snapshot metadata and data will consistently reflect the latest snapshot that Retro has declared and the last pre-state it has saved before the crash. In Stage 2 therefore, Line 10 will suppress a duplicate snapshot declaration (A later snapshot is already present in Start). Line 17 will suppress duplicate saving of a pre-state (SavedAfter indicates the pre-state has been already saved).

We have considered a simplified writing policy that only writes database pages and Retro metadata at checkpoint time. Retro also supports more flexible writing policies. In particular, if the database is forced to write database pages between checkpoints due to cache pressure, Retro can still enforce its writing invariants by writing the pre-states of the pages forced from the cache (and making durable the associated snapshot metadata), and recover correctly if there is a crash before the next checkpoint. Suppression will still work correctly in this case because it is applied per-page based on a lookup in

Algorithm 2 Retro recovery

```
Recover snapshot metadata                                   ▷ begin Stage 1
Delete unreferenced pre-states from Pagelog
$S_{latest}$ := latest snapshot id in Start                 ▷ begin Stage 2
Normal database recovery begins replay
repeat
    if Recovery requests page $P$ for updating from the cache then
        Make a copy of the pre-state of $P$ in the cache
    end if
    if Recovery encounters a snaprec for snapshot $S$ then
        if $S > S_{latest}$ then
            Declare $S$
            $S_{latest} := S$
        end if
    end if
    if Recovery encounters a commit record for transaction $T$ then
        for all Pre-states $P$ created by $T$ do
            if $SavedAfter(P) < S_{latest}$ then
                Save $P$ for $S_{latest}$
            end if
        end for
    end if
until REDO recovery is complete
Checkpoint database                   ▷ Retro will enforce the WAS invariant
```

SavedAfter, just like during normal operation.

7 Implementation Issues

Retro protocol extensions are implemented as a set of callbacks invoked from BDB transaction commit, recovery, and buffer cache protocols. There are two concerns in the way of an efficient implementation of the protocol extensions. Extensions that run concurrently, accessing shared mutable state, must be thread safe, and must be serialized in transaction order to correctly save and access snapshots. However, the implementation needs to avoid blocking transactions to synchronize extensions. Second, although storing Retro metadata in transactional structures simplifies snapshot recovery, frequent updates to metadata are costly. So, the implementation needs to reduce the cost of metadata updates.

7.1 Latest Snapshot

Extensions that declare snapshots, and save pre-states and metadata, invoked at commit, need to read and update the latest snapshot id. These extensions need to be serialized in commit order to correctly assign snapshot ids (numbered sequentially in declaration commit order), and to correctly identify updates tagged in OWS (is this the first update to P following the latest snapshot declaration?).

Extensions that create persistent data (e.g. save pre-states) must run after the invoking transaction commits (i.e. records its commit record in the log). The problem arises because concurrent transaction threads that block to write their commit records, get unblocked by BDB in arbitrary order (possibly after BDB runs a group commit), thus reordering the execution of extensions.

An extension must therefore determine the latest snapshot declaration preceding its transaction regardless of extension execution order. Instead of tracking the latest snapshot in a shared counter, Retro solves the problem by tracking recent snapshot declarations in a data structure called the SnapshotList. Entries for transactions that have completed the log write and therefore were assigned a commit LSN are sorted by their commit LSN. Retro uses the SnapshotList to assign snapshot ids sequentially in transaction commit LSN order regardless of extension execution order. An extension can determine the last-declared snapshot S_{last} for its invoking transaction T using the SnapshotList. It simply searches for the snapshot declaration with the highest LSN preceding the commit LSN of T.

7.2 SavedAfter cache

After saving a pre-state, Retro needs to update SavedAfter, to note it has been saved. Updating SavedAfter is expensive because it is a transactional data structure, so we describe a specialized write cache called the SavedAfter cache (SAC) that allows deferring updates to SavedAfter and consistent checking of SavedAfer.

In the Retro recovery protocol, Retro metadata is recovered first, before the application database, which means that the log into which SavedAfter updates are recorded must be separate from the BDB transaction log. As a consequence, when the commit extension needs to update SavedAfter(P), updating the durable SavedAfter metadata structure would require a second commit and log write, in addition to the commit of the application's transaction. We have measured the impact of committing an update to SavedAfter after saving a pre-state in commit extension and unsurprisingly, it has a significant impact on transaction throughput and latency.

To reduce the impact, we introduce an in-memory cache, called the SavedAfter cache (SAC), to store updates to SavedAfter on cache pages. Updates are propagated from SAC to SavedAfter when Retro data and metadata are flushed to the Retro disk. SAC therefore eliminates the SaveAfter update cost from the transaction commit path, allowing multiple SavedAfter updates to be combined into a into a single transaction, and multiple updates to the same entry (e.g., SavedAfter?) to be absorbed.

Because the entries in SAC may be newer than those in SavedAfter, Retro must check SAC when looking up SavedAfter(P). Just as with SavedAfter, the SAC is a frequently-accessed data structure. However, SAC is not a source of extra contention in the system, because it leverages MVCC page-level locking and low-level concurrency mechanisms in the page cache.

SAC is implemented by extending the BDB page cache structures. A page P cached in the BDB page cache is preceded by a page header that contains meta information about the page such as its name, the commit LSN of the last transaction to update the page, the linked list of pages that form the version chain (VC) for this page, maintained by MVCC, and other internal metadata. SAC(*P*) is stored on the header for cached page P (requiring that every page header be enlarged by the size of a snapshot id).

SAC(*P*) is initialized from SavedAfterCache(P) when there is a cache miss and P is read from disk. When MVCC creates a new version of page P in the cache for an update to P, SAC of the new version is initialized by the commit extension of the transaction that commits the update. When checking whether to save the pre-state, the commit extension uses the SAC value on the pre-state. If the pre-state of P needs to be saved for snapshot S, the SAC on the new version of P will be set to S. Otherwise, the SAC on the new version of P will be copied from the pre-state of P.

7.3 SAC and Retrospection

Retro runs a retrospective query as MVCC transaction T. An extension invoked from T observes the Vsnap of T. Consider a retrospective query T as off snapshot S, serialized after transaction execution H. A page translation extension, invoked when T accesses a page P, will observe a consistent SAC(*P*) value as of the begin LSN of T, reflecting all tagged update commits and effects of their associated extensions that precede T in OWS(H). If the SAC value indicates P@S is shared with the database, (i.e. it was shared when T began) the translation will correctly direct the access to the version of P from *Vsnap* of T. Otherwise the translation will redirect the access to *P@S* saved by Retro, as required by OWS specification.

8 Performance Evaluation

Our simple study evaluates the performance of retrospection, the new feature, and Retro overheads to BDB transactions. The results confirm the low overhead of Retro to BDB. The results also show a slowdown for running Retro transactions (i.e., transactions that use snapshots); reducing these overheads is an area for future research.

The Retro prototype extends BDB version 5.3.21. The prototype has just over 5000 lines of C code. The modifications to BDB to integrate Retro are minimal: under 250 LOC were modified or added to BDB source code. The implementation includes the complete design for the C API used in the evaluation. We are in the process of completing the SQLite API. Preliminary results using SQLite API confirm the results from the C API.

The hardware platform is a quad-core Intel Xeon CPU at 2.66ghz with 4 gb of physical RAM and 2 Seagate Cheetah 15,500 RPM SAS hard drives, running DEBIAN GNU/Linux version 2.6.32. BDB stores database files in the file system, formatted with ext3. BDB uses default page size of 4K. Disk level prefetching is disabled, to emphasize the cost of random disk I/O for Pagelog. Our platform only supports two disks, so we use one for the database, Pagelog, and metadata, and one for the (separate) database and metadata logs (BDB database and log must be on separate disks since otherwise BDB (without Retro) transaction throughput drops to zero during a checkpoint). This is sub-optimal configuration for Retro, since Pagelog could impact BDB reads and writes. Our experiments use in-memory working sets for the database, insulating database reads from Pagelog I/O. Moreover, since Retro trickles snapshots to disk between checkpoints (unless noted otherwise), this is not a problem for database writes in our experiments.

All measurements are reported as the average of 10 runs (non-negligible standard deviations are depicted).

8.1 Retrospection

Page name translation. We evaluate the performance of Retro transactions using a simple read query *Q* that fetches 2048 random records (each 108 bytes in size) from a table cached in memory. We report *slow-down*, the measured time-to-completion relative to running *Q* in-memory in unmodified BDB. The workload maximizes the CPU overhead of Retro transactions since *Q* performs no computation.

The CPU overhead of page name translation comes from looking up the page in SavedAfter (the current state page is not cached, no SAC) or SAC and, if the snapshot page has been already saved as a pre-state, looking up the location of that pre-state in SPT.

Figure 5 shows that a slow-down when accessing pages shared with the current state is 1.6x ("Retro(Q) cur") using SAC. This overhead is similar in magnitude and origin to the one that was found due to the buffer pool indirection and latching for tree lookups in Shore [2]. The slow-down when accessing snapshot pages saved in

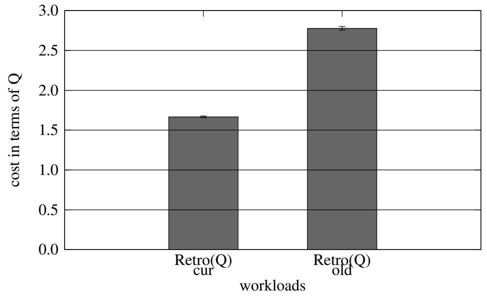


Figure 5: CPU costs

Component	*Cost*
SPT(S) get	0.55*us*
SavedAfter get	47*us*
SAC get	0.06 *us*
SPT(S) put	11*us*
Pagelog get	3.62 *ms*

Figure 6: Translation costs

Pagelog ("Retro(Q) old") is about a 2.7x, due to additional access to SPT, including lookup and insertion of mappings into the SPT by scanning Maplog. The slowdown when accessing Pagelog is higher when the pages accessed by retrospection do not have a current state version cached in memory (no SAC), and therefore a costlier lookup in SavedAfter is needed. The dominant cost however is accessing Pagelog.

The absolute costs of translation components are shown in Table 6. Insert into the SPT is costly, more so than lookup. The SPT is implemented using a simple hash table that is not optimized for resizing, and resizing is frequent during Maplog scan. Maplog in-memory scan is costly. Maplog is composed of many small mappings that must be individually searched. Maplog is not shown in the table because it does not have a clear per request cost. The total contribution of costs from Maplog resembles total contribution of SPT insert. Optimizing access to SPT and Pagelog is an area of future work.

Snapshot page I/O. I/O from the Pagelog has different performance characteristics than I/O from the current state database. Copy-on-write declusters snapshot pages, resulting in a different spatial layout from the current state. A sequential query incurring sequential disk I/O costs in the current state can perform poorly when running retrospectively, incurring random disk I/O costs, similar to I/O in a log structured file system [11]). Large memories and SSDs can eliminate the declustering penalty, and we plan to use SSD for Retro in future work. Nevertheless, for large volumes of snapshots, the SSD solution may not be practical. Since declustering effect for sequential queries is well understood, here we focus on random queries to expose an additional source of Pagelog performance difference not directly related to declustering. When the update workload is non-uniform (skewed), pages from the same snapshot may be very far apart. By modeling the skew in the update workload, we can characterize the I/O overhead of random queries run retrospectively. We use a standard model of skewed workload: "80/20" means 80% of the updates go to 20% of the database, "50/50" means the workloads is not skewed. The portion of Pagelog corresponding to any snapshot fits on a single disk; the seek impact may actually be mitigated if Pagelog is spread over multiple disks.

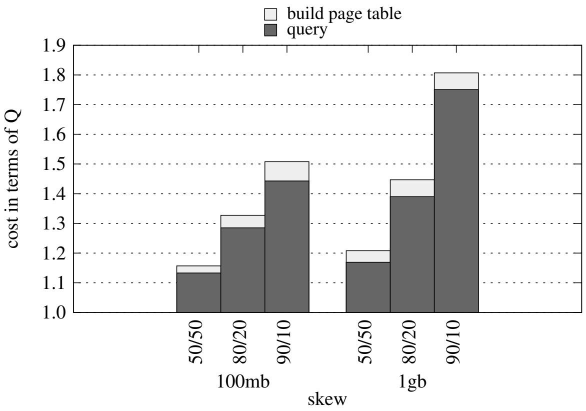


Figure 7: I/O-bound retro query costs

Figure 7 compares the cost of I/O for $Retro(Q)$ for different update skews to the cost of I/O for Q in BDB. The experiment runs with cold caches. The query for this experiment is identical to the "Retro(Q) old" query depicted in figure 5, except that the cache starts cold (we report "slow-down" due to Retro as a ratio; the absolute latencies for I/O-bound queries were 3 to 4 orders higher than in the CPU-bound experiments). We include breakdown for snapshot data and metadata. The full analysis of Skippy index appears in [14].

The skew impact persists independently of table size. We run the experiment with 100MB table and a 1GB table (figure 7). We scaled up the number of random records read in the query for the larger table so that in both cases the query reads 1% of the table. For both the small and large tables, there is a similar trend of increasing cost of $Retro(Q)$ relative to Q as skew increases. The impact of skew appears higher in the larger database, but this is due to the decreased hit ratio in the larger query. In the large table Q had a 7% hit ratio in the leaf pages of the table, as opposed to 14% in the small table (the cache starts cold, a single page read pre-fetches multiple records, and given a constant page size, the likelihood of a subsequent hit on the pre-fetched page decreases as the table size increases because a smaller fraction of table records are clustered on each page).

8.2 Overhead to BDB

Retro adds no direct overhead to BDB queries. To updates, Retro adds commit time CPU overhead (SAC must be checked and possibly updated for each update), and possibly synchronous I/O (to log a snapshot declaration record). Enforcing the Before-GC invariant (section 5) requires writing pre-states in the background but does not delay commits. Enforcing the WAS invariant (Sec 6) can increase checkpoint latency due to snapshot I/O. Trickling snapshots between checkpoints avoids the checkpoint delay. Writing snapshots to a different disk avoids contention between Retro and the current state during checkpoints altogether. Our system has two disks: one is used for the transaction log, so Pagelog and current state share a disk.

We run a simple micro benchmark to test the overhead to update transactions running in memory incurred by saving snapshot pre-states in-memory and writing all snapshot data and metadata to disk during checkpoints. We ran the random update transaction described in section 8.1 in Retro, and unmodified BDB. With Retro, the system declares a snapshot after each update transaction, saving pre-states for the pages modified by 1000 random updates on each transaction commit, thus maximizing the number of snapshot pages that must be saved. We calculate throughput from the time-to-completion and the number of completed update transactions. We observe an overhead of about 4% to update throughput in our workload from Retro, confirming previous results [15] concerning the low impact of the split snapshot writes.

9 Conclusion

We described Retro, a new efficient system for retrospection in Berkeley DB, implemented using a new simple and robust method that avoids invasive database modifications. Our approach is adapted to BDB. However, because WAL, MVCC and buffer cache are standard protocols, we believe the approach is more general and we are investigating other extensions in on-going work. The key challenges going forward are optimizing the performance of retrospective queries and supporting SSDs.

References

[1] BRIDGE, W., JOSHI, A., KEIHL, M., LAHIRI, T., LOAIZA, J., AND MACNAUGHTON, N. The Oracle universal server buffer. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1997), Morgan Kaufmann Publishers Inc., pp. 590–594.

[2] HARIZOPOULOS, S., ABADI, D. J., MADDEN, S., AND STONEBRAKER, M. OLTP through the looking glass, and what we found there. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 981–992.

[3] HELLERSTEIN, J. M., STONEBRAKER, M., AND HAMILTON, J. Architecture of a database system. In *Foundations and Trends in Databases* (2007), vol. 1.

[4] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an nfs file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 19–19.

[5] LOMET, D., BARGA, R., MOKBEL, M. F., SHEGALOV, G., WANG, R., AND ZHU, Y. Immortal DB: transaction time support for SQL server. In *Proceedings of the ACM SIGMOD international conference on Management of data* (2005), pp. 939–941.

[6] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst. 17* (March 1992), 94–162.

[7] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A. P., AND ZADOK, E. A versatile and user-oriented versioning file system. In *FAST* (2004).

[8] ORACLE CORPORATION. Berkeley DB degrees of isolation. `http://www.oracle.com/technology/documentation/berkeley-db/db/programmer_reference/transapp_read.html`.

[9] OZSOYOGLU, G., AND SNODGRASS, R. T. Temporal and real-time databases: A survey. *Knowledge and Data Engineering 7*, 4 (1995).

[10] PLATTNER, C., WAPF, A., AND ALONSO, G. Searching in time. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2006), ACM, pp. 754–756.

[11] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[12] SANTRY, D., FEELEY, M., HUTCHINSON, N., VEITCH, A., CARTON, R., AND OTIR, J. Deciding when to forget in the elephant file system. In *Symposium on Operating Systems Principles* (1999).

[13] SHAULL, R. *Retro: A methodology for Retrospection Everywhere*. PhD thesis, Brandeis University, Aug. 2013.

[14] SHAULL, R., SHRIRA, L., AND XU, H. Skippy: a new snapshot indexing method for time travel in the storage manager. In *SIGMOD Conference* (2008).

[15] SHRIRA, L., AND XU, H. Snap: Efficient snapshots for back-in-time execution. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering* (Washington, DC, USA, 2005), IEEE Computer Society.

[16] SHRIRA, L., AND XU, H. Thresher: An efficient storage manager for copy-on-write snapshots. In *USENIX '06: Proceedings* (Berkeley, CA, USA, 2006), Advanced Computer Systems Association.

[17] WIRES, J., AND FEELEY, M. J. Secure file system versioning at the block level. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), ACM, pp. 203–215.

SCFS: A Shared Cloud-backed File System

Alysson Bessani,[1] *Ricardo Mendes*,[1] *Tiago Oliveira*,[1] *Nuno Neves*,[1]
Miguel Correia,[2] *Marcelo Pasin*,[3] *Paulo Verissimo*[1]
[1]Faculdade de Ciências/LaSIGE, [2]Instituto Superior Técnico/INESC-ID, Universidade de Lisboa, Portugal
[3]University of Neuchatel, Switzerland

Abstract

Despite of their rising popularity, current cloud storage services and cloud-backed storage systems still have some limitations related to reliability, durability assurances and inefficient file sharing. We present SCFS, a cloud-backed file system that addresses these issues and provides strong consistency and near-POSIX semantics on top of eventually-consistent cloud storage services. SCFS provides a pluggable backplane that allows it to work with various storage clouds or a cloud-of-clouds (for added dependability). It also exploits some design opportunities inherent in the current cloud services through a set of novel ideas for cloud-backed file systems: always write and avoid reading, modular coordination, private name spaces and consistency anchors.

1 Introduction

File backup, data archival and collaboration are among the top usages of the cloud in companies [1], and they are normally based on cloud storage services like the Amazon S3, Dropbox, Google Drive and Microsoft SkyDrive. These services are popular because of their ubiquitous accessibility, pay-as-you-go model, high scalability, and ease of use. A cloud storage service can be accessed in a convenient way with a client application that interfaces the local file system and the cloud. Such services can be broadly grouped in two classes: (1) personal file synchronization services (e.g., DropBox) and (2) cloud-backed file systems (e.g., S3FS [6]).

Services of the first class – personal file synchronization – are usually composed of a back-end storage cloud and a client application that interacts with the local file system through a monitoring interface like *inotify* (in Linux). Recent works show that this interaction model can lead to reliability and consistency problems on the stored data [41], as well as CPU and bandwidth over usage under certain workloads [35]. In particular, given the fact that these monitoring components lack an understanding of when data or metadata is made persistent in the local storage, this can lead to corrupted data being saved in the cloud. A possible solution to these difficulties would be to modify the file system to increase the integration between client application and local storage.

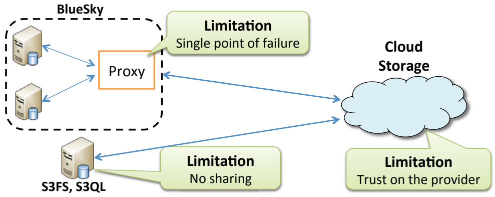


Figure 1: Cloud-backed file systems and their limitations.

The second class of services – cloud-backed file systems – solves the problem in a more generic way. This approach is typically implemented at user-level, following one of the two architectural models represented in Figure 1. The first model is shown at the top of the figure and is followed by BlueSky [39] and several commercial storage gateways. In this model, a proxy component is placed in the network infrastructure of the organization, acting as a file server to multiple clients and supporting access protocols such as NFS and CIFS. The proxy implements the core file system functionality and calls the cloud to store and retrieve files. The main limitations are that the proxy can become a performance bottleneck and a single point of failure. Moreover, in BlueSky (and some other systems) there is no coordination between different proxies accessing the same files. The second model is implemented by open-source solutions like S3FS [6] and S3QL [7] (bottom of Figure 1). In this model, clients access the cloud directly, without the interposition of a proxy. Consequently, there is no longer a single point of failure, but the model misses the convenient rendezvous point for synchronization, making it harder to support controlled file sharing among clients.

A common limitation of the two classes of services is the need to trust the cloud provider with respect to the stored data confidentiality, integrity and availability. Although confidentiality can be guaranteed by making clients (or the proxy) encrypt files before sending them to the cloud, sharing encrypted files requires a key distribution mechanism, which is not easy to implement in this environment. Integrity is provided by systems like SUNDR [34], which requires the execution of specific code on the cloud provider, currently not possible when using unmodified storage services. Availability against cloud failures to the best of our knowledge is not provided by any of the current cloud-backed file systems.

This paper presents the *Shared Cloud-backed File System (SCFS)*,[1] a storage solution that addresses the aforementioned limitations. SCFS allows entities to *share* files in a secure and fault-tolerant way, improving the durability guarantees. It also ensures strong consistency on file accesses, and provides a backplane that can plug on multiple different cloud storage services.

SCFS leverages almost 30 years of distributed file systems research, integrating classical ideas like consistency-on-close semantics [28] and separation of data and metadata [21], with recent trends such as using cloud services as (unmodified) storage backends [20, 39] and increasing dependability by resorting to multiple clouds [9, 12, 15]. These ideas were augmented with the following *novel techniques* for cloud-backed storage design:

- *Always write / avoid reading:* SCFS always pushes updates of file contents to the cloud (besides storing them locally), but resolves reads locally whenever possible. This mechanism has a positive impact in the reading latency. Moreover, it reduces costs because writing to the cloud is typically cheap, on the contrary of reading that tends to be expensive.[2]
- *Modular coordination:* SCFS uses a fault-tolerant coordination service, instead of an embedded lock and metadata manager, as most distributed file systems [10, 32, 40]. This service has the benefit of assisting the management of consistency and sharing. Moreover, the associated modularity is important to support different fault tolerance tradeoffs.
- *Private Name Spaces:* SCFS uses a new data structure to store metadata information about files that are not shared between users (which is expected to be the majority [33]) as a single object in the storage cloud. This relieves the coordination service from maintaining information about such private files and improves the performance of the system.
- *Consistency anchors:* SCFS employs this novel mechanism to achieve strong consistency, instead of the eventual consistency [38] offered by most cloud storage services, a model typically considered unnatural by a majority of programmers. This mechanism provides a familiar abstraction – a file system – without requiring modifications to cloud services.
- *Multiple redundant cloud backends:* SCFS may employ a cloud-of-clouds backplane [15], making the system tolerant to data corruption and unavailability of cloud providers. All data stored in the clouds is encrypted for confidentiality and encoded for storage-efficiency.

The use case scenarios of SCFS include both individuals and large organizations, which are willing to explore the benefits of cloud-backed storage (optionally, with a cloud-of-clouds backend). For example: *a secure personal file system* – similar to Dropbox, iClouds or SkyDrive, but without requiring complete trust on any single provider; *a shared file system for organizations* – cost-effective storage, but maintaining control and confidentiality of the organizations' data; *an automatic disaster recovery system* – the files are stored by SCFS in a cloud-of-clouds backend to survive disasters not only in the local IT systems but also of individual cloud providers; *a collaboration infrastructure* – dependable data-based collaborative applications without running code in the cloud, made easy by the POSIX-like API for sharing files.

Despite the fact that distributed file systems are a well-studied subject, our work relates to an area where further investigation is required – cloud-backed file systems – and where the practice is still somewhat immature. In this sense, besides presenting a system that explores a novel region of the cloud storage design space, the paper contributes with a set of generic principles for cloud-backed file system design, reusable in further systems with different purposes than ours.

2 SCFS Design

2.1 Design Principles

This section presents a set of design principles that are followed in SCFS:

Pay-per-ownership. Ideally, a shared cloud-backed file system should charge each owner of an account for the files it creates in the service. This principle is important because it leads to a flexible usage model, e.g., allowing different organizations to share directories paying only for the files they create. SCFS implements this principle by reusing the protection and isolation between different accounts granted by the cloud providers (see §2.6).

Strong consistency. A file system is a more familiar storage abstraction to programmers than the typical basic interfaces (e.g., REST-based) given by cloud storage services. However, to emulate the semantics of a POSIX file system, strong consistency has to be provided. SCFS follows this principle by applying the concept of consistency anchors (see §2.4). Nevertheless, SCFS optionally supports weaker consistency.

Service-agnosticism. A cloud-backed file system should rule out from its design any feature that is not supported by the backend cloud(s). The importance of this principle derives from the difficulty (or impossibility) in obtaining modifications of the service of the best-of-breed commercial clouds. Accordingly, SCFS does not assume any special feature of storage clouds besides on-demand access to storage and basic access control lists.

[1] SCFS is available at http://code.google.com/p/depsky/wiki/SCFS.

[2] For example, in Amazon S3, writing is free, but reading a GB is more expensive ($0.12 after the first GB/month) than storing data during a month ($0.09 per GB). Google Cloud Storage's prices are similar.

Multi-versioning. A shared cloud-backed file system should be able to store several versions of the files for error recovery [23]. An important advantage of having a cloud as backend is its potentially unlimited capacity and scalability. SCFS keeps old versions of files and deleted files until they are definitively removed by a configurable garbage collector.

2.2 Goals

A primary goal of SCFS is to allow clients to share files in a controlled way, providing the necessary mechanisms to guarantee security (integrity and confidentiality; availability despite cloud failures is optional). An equally important goal is to increase data durability by exploiting the resources granted by storage clouds and keeping several versions of files.

SCFS also aims to offer a natural file system API with strong consistency. More specifically, SCFS supports consistency-on-close semantics [28], guaranteeing that when a file is closed by a user, all updates it saw or did are observable by the rest of the users. Since most storage clouds provide only eventual consistency, we resort to a coordination service [13, 29] for maintaining file system metadata and synchronization.

A last goal is to leverage the clouds' services scalability, supporting large numbers of users and files as well as large data volumes. However, SCFS is not intended to be a big-data file system, since file data is uploaded to and downloaded from one or more clouds. On the contrary, a common principle for big-data processing is to take computation to the data (e.g., MapReduce systems).

2.3 Architecture Overview

Figure 2 represents the SCFS architecture with its three main components: the *backend cloud storage* for maintaining the file data (shown as a cloud-of-clouds, but a single cloud can be used); the *coordination service* for managing the metadata and to support synchronization; and the *SCFS Agent* that implements most of the SCFS functionality, and corresponds to the file system client mounted at the user machine.

The separation of file data and metadata has been often used to allow parallel access to files in parallel file systems (e.g., [21, 40]). In SCFS we take this concept further and apply it to a cloud-backed file system. The fact that a distinct service is used for storing metadata gives flexibility, as it can be deployed in different ways depending on the users needs. For instance, our general architecture assumes that metadata is kept in the cloud, but a large organization could distribute the metadata service over its own sites for disaster tolerance.

Metadata in SCFS is stored in a coordination service. Three important reasons led us to select this approach instead of, for example, a NoSQL database or some custom service (as in other file systems). First, coordination services offer *consistent* storage with enough capacity for this kind of data, and thus can be used as *consistency anchors* for cloud storage services (see next section). Second, coordination services implement complex replication protocols to ensure *fault tolerance* for metadata storage. Finally, these systems support operations with *synchronization* power [26] that can be used to implement fundamental file system functionalities, such as locking.

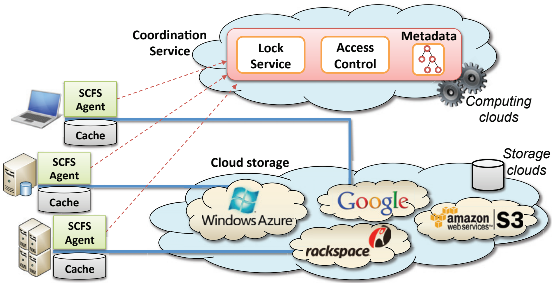


Figure 2: SCFS architecture.

File data is maintained both in the storage cloud and locally in a cache at the client machine. This strategy is interesting in terms of performance, costs and availability. As cloud accesses usually entail large latencies, SCFS attempts to keep a copy of the accessed files in the user's machine. Therefore, if the file is not modified by another client, subsequent reads do not need to fetch the data from the clouds. As a side effect, there are cost savings as there is no need to pay to download the file. On the other hand, we follow the approach of writing everything to the cloud, as most providers let clients upload files for free as an incentive to use their services. Consequently, no completed update is lost in case of a local failure.

According to our design, the storage cloud(s) and the coordination service are external services. SCFS can use any implementation of such services as long as they are compatible (provide compliant interfaces, access control and the required consistency). We will focus the rest of this section on the description of the SCFS Agent and its operation principles, starting with how it implements consistent storage using weakly consistent storage clouds.

2.4 Strengthening Cloud Consistency

A key innovation of SCFS is the ability to provide strongly consistent storage over the eventually-consistent services offered by clouds [38]. Given the recent interest in strengthening eventual consistency in other areas, we describe the general technique here, decoupled from the file system design. A complete formalization and correctness proof of this technique is presented in a companion technical report [14].

The approach uses two storage systems, one with limited capacity for maintaining metadata and another to save the data itself. We call the metadata store a *consistency anchor* (CA) and require it to enforce some desired consistency guarantee S (e.g., linearizability [27]), while the

```
WRITE($id, v$):
w1: $h \leftarrow \mathsf{Hash}(v)$
w2: SS.write($id|h, v$)
w3: CA.write($id, h$)

READ($id$):
r1: $h \leftarrow$ CA.read($id$)
r2: do $v \leftarrow$ SS.read($id|h$) while $v = null$
r3: return $(\mathsf{Hash}(v) = h)?v : null$
```

Figure 3: Algorithm for increasing the consistency of the storage service (SS) using a consistency anchor (CA).

storage service (SS) may only offer eventual consistency. The objective is to provide a composite storage system that satisfies S, even if the data is kept in SS.

The algorithm for improving consistency is presented in Figure 3, and the insight is to anchor the consistency of the resulting storage service on the consistency offered by the CA. For writing, the client starts by calculating a collision-resistant hash of the data object (step w1), and then saves the data in the SS together with its identifier id concatenated with the hash (step w2). Finally, data's identifier and hash are stored in the CA (step w3). Every write operation creates a new version of the data object and garbage collection is required to reclaim the storage space of no longer needed versions.

For reading, the client has to obtain the current hash of the data from CA (step r1), and then needs to keep on fetching the data object from the SS until a copy is available (step r2). The loop is necessary due to the eventual consistency of the SS – after a write completes, the new hash can be immediately acquired from the CA, but the data is only eventually available in the SS.

2.5 SCFS Agent

2.5.1 Local Services

The design of the SCFS Agent is based on the use of three *local* services that abstract the access to the coordination service and the storage cloud backend.

Storage service. The storage service provides an interface to save and retrieve variable-sized objects from the cloud storage. SCFS overall performance is heavily affected by the latency of remote (Internet) cloud accesses. To address this problem, we read and write whole files as objects in the cloud, instead of splitting them in blocks and accessing block by block. This allows most of the client files (if not all) to be stored locally, and makes the design of SCFS simpler and more efficient for small-to-medium sized files.

To achieve adequate performance, we rely on two levels of cache, whose organization has to be managed with care in order to avoid impairing consistency. First, all files read and written are copied locally, making the local disk a large and long-term cache. More specifically, the disk is seen as an LRU file cache with GBs of space, whose content is validated in the coordination service before being returned, to ensure that the most recent version of the file is used. Second, a main memory LRU cache (hundreds of MBs) is employed for holding open files. This is aligned with our consistency-on-close semantics, since, when the file is closed, all updated metadata and data kept in memory are flushed to the local disk and the clouds.

Actual data transfers between the various storage locations (memory, disk, clouds) are defined by the durability levels required by each type of system call. Table 1 shows examples of POSIX calls that cause data to be stored at different levels, together with their location, storage latency and fault tolerance. For instance, a write in an open file is stored in the memory cache, which gives no durability guarantees (Level 0). Calling `fsync` flushes the file (if modified) to the local disk, achieving the standard durability of local file systems, i.e., against process or system crashes (Level 1). When a file is closed, it is eventually written to the cloud. A system backed by a single cloud provider can survive a local disk failure but not a cloud provider failure (Level 2). However, in SCFS with a cloud-of-clouds backend, files are written to a set of clouds, such that failure of up to f providers is tolerated (Level 3), being f a system parameter (see §3.2).

Level	*Location*	*Latency*	*Fault tol.*	*Sys. call*
0	main memory	microsec	none	write
1	local disk	millisec	crash	fsync
2	cloud	seconds	local disk	close
3	cloud-of-clouds[1]	seconds	f clouds	close

Table 1: SCFS durability levels and the corresponding data location, write latency, fault tolerance and example system calls. [1] Supported by SCFS with the cloud-of-clouds backend.

Metadata service. The metadata service resorts to the coordination service to store file and directory metadata, together with information required for enforcing access control. Each file system object is represented in the coordination service by a metadata tuple containing: the object name, the type (file, directory or link), its parent object (in the hierarchical file namespace), the object metadata (size, date of creation, owner, ACLs, etc.), an opaque identifier referencing the file in the storage service (and, consequently, in the storage cloud) and the collision-resistant hash (SHA-1) of the contents of the current version of the file. These two last fields represent the id and the $hash$ stored in the consistency anchor (see §2.4). Metadata tuples are accessed through a set of operations offered by the local metadata service, which are then translated into different calls to the coordination service.

Most application actions and system call invocations are translated to several metadata accesses at the file system level (e.g., opening a file with the `vim` editor can cause more than five `stat` calls for the file). To deal with these access bursts, a small short-term metadata cache is kept in main memory (up to few MBs for tens of milliseconds). The objective of this cache is to reuse the data fetched from the coordination service for at least the amount of time spent to obtain it from the network.

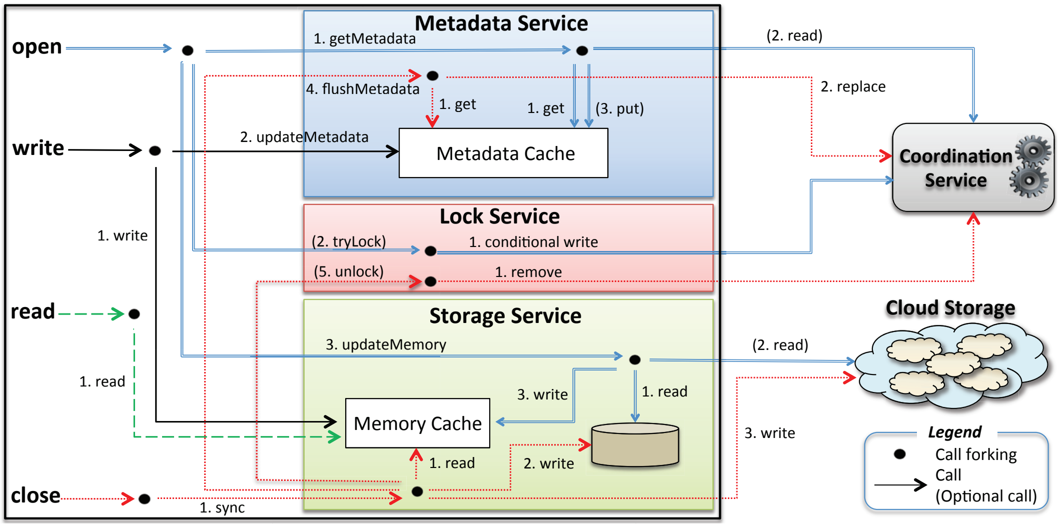


Figure 4: Common file system operations in SCFS. The following conventions are used: 1) at each call forking (the dots between arrows), the numbers indicate the order of execution of the operations; 2) operations between brackets are optional; 3) each file system operation (e.g., open/close) has a different line pattern.

Notice that accessing cached metadata can lead to violations of strong consistency. For this reason, we maintain such cached information for very short time periods, only to serve the file system calls originated from the same high-level action over a file (e.g., opening or saving a document). In §4.4 we show that this cache significantly boosts the performance of the system.

Locking service. As in most consistent file systems, we use *locks* to avoid write-write conflicts. The lock service is basically a wrapper for implementing coordination recipes for locking using the coordination service of choice [13, 29]. The only strict requirement is that the lock entries inserted are ephemeral. In practice, locks can be represented by ephemeral znodes in Zookeeper or timed tuples in DepSpace, ensuring they will disappear (automatically unlocking the file) in case a SCFS client that locked a file crashes before uploading its updates and releasing the lock (see next section).

Opening a file for reading does not require locking it. Read-write conflicts are automatically addressed when uploading and downloading whole files and using consistency anchors (see §2.4) which ensure the most recent version of file (according to consistency-on-close) will be read upon its opening.

2.5.2 File Operations

Figure 4 illustrates the execution of the four main file system calls (open, write, read and close) in SCFS.

Opening a file. The tension between provisioning strong consistency and suffering high latency in cloud access led us to provide consistency-on-close semantics [28] and synchronize files only in the open and close operations. Moreover, given our aim of having most client files (if not all) locally stored, we opted for reading and writing whole files from the cloud. With this in mind, the open operation comprises three main steps: (i) read the file metadata, (ii) optionally create a lock if the file is opened for writing, and (iii) read the file data to the local cache. Notice that these steps correspond to an implementation of the *read* algorithm of Figure 3, with an extra step to ensure exclusive access to the file for writing.

Reading the metadata entails fetching the file metadata from the coordination service, if it is not available in the metadata cache, and then make an update to this cache. Locking the file is necessary to avoid write-write conflicts, and if it fails, an error is returned. File reads are either done in the local cache (memory or disk) or in the cloud. The local file version (if available) is compared with the version stored in the metadata service. If a newer version exists, it is read from the cloud and cached in the local disk and in main memory. If there is no space for the file in main memory (e.g., there are too many open files), the data of the least recently used file is first pushed to disk (as a cache extension) to release space.

Write and read. These two operations only need to interact with the local storage. Writing to a file requires updating the memory-cached file and the associated metadata cache entry (e.g., the size and the last-modified timestamp). Reading just causes the data to be fetched from the main memory cache (as it was copied there when the file was opened).

Closing a file. Closing a file involves the synchronization of cached data and metadata with the coordination service and the cloud storage. First, the updated file data is copied to the local disk and to the storage cloud. Then, if the cached metadata was modified, it is pushed to the coordination service. Lastly, the file is unlocked if it was originally opened for writing. Notice that these steps correspond to the *write* algorithm of Figure 3.

As expected, if the file was not modified since opened or was opened in read-only mode, no synchronization is required. From the point of view of consistency and durability, a write to the file is complete only when the file is closed, respecting the consistency-on-close semantics.

2.5.3 Garbage Collection

During normal operation, SCFS saves new file versions without deleting the previous ones, and files removed by the user are just marked as deleted in the associated metadata. These two features support the recovery of old versions of the files, which is useful for some applications. Keeping old versions of files increases storage costs, and therefore, SCFS includes a flexible garbage collector to enable various policies for reclaiming space.

Garbage collection runs in isolation at each SCFS Agent, and the decision about reclaiming space is based on the preferences (and budgets) of individual users. By default, its activation is guided by two parameters defined upon the mounting of the file system: *number of written bytes W* and *number of versions to keep V*. Every time an SCFS Agent writes more than W bytes, it starts the garbage collector as a separated thread that runs in parallel with the rest of the system (other policies are possible). This thread fetches the list of files owned by this user and reads the associated metadata from the coordination service. Next, it issues commands to delete old file data versions from the cloud storage, such that only the last V versions are kept (refined policies that keep one version per day or week are also possible). Additionally, it also eliminates the versions removed by the user. Later on, the corresponding metadata entries are also erased from the coordination service.

2.6 Security Model

The security of a shared cloud storage system is a tricky issue, as the system is constrained by the access control capabilities of the backend clouds. A straw-man implementation would allow all clients to use the same account and privileges on the cloud services, but this has two drawbacks. First, any client would be able to modify or delete all files, making the system vulnerable to malicious users. Second, a single account would be charged for all clients, preventing the pay-per-ownership model.

Instead of classical Unix modes (*owner*, *group*, *others*; *read*, *write*, *execute*), SCFS implements ACLs [22]. The owner O of a file can give access permissions to another user U through the `setfacl` call, passing as parameters file name, identifier of user U, and permissions. Similarly, `getfacl` retrieves the permissions of a file.

As a user has separate accounts in the various cloud providers, and since each probably has a different identifier, SCFS needs to associate with every client a list of cloud canonical identifiers. This association is kept in a tuple in the coordination service, and is loaded when the client mounts the file system for the first time. When the SCFS Agent intercepts a `setfacl` request from a client O to set permissions on a file for a user U, the following steps are executed: (i) the agent uses the two lists of cloud canonical identifiers (of O and U) to update the ACLs of the objects that store the file data in the clouds with the new permissions; and then, (ii) it also updates the ACL associated with the metadata tuple of the file in the coordination service to reflect the new permissions.

Notice that we do not trust the SCFS Agent to implement the access control verification, since it can be compromised by a malicious user. Instead, we rely on the access control enforcement of the coordination service and the cloud storage.

2.7 Private Name Spaces

One of the goals of SCFS is to scale in terms of users and files. However, the use of a coordination service (or any centralized service) could potentially create a scalability bottleneck, as this kind of service normally maintains all data in main memory [13, 29] and requires a distributed agreement to update the state of the replicas in a consistent way. To address this problem, we take advantage of the observation that, although file sharing is an important feature of cloud-backed storage systems, the majority of the files are not shared between different users [20, 33]. Looking at the SCFS design, all files and directories that are not shared (and thus not visible to other users) do not require a specific entry in the coordination service, and instead can have their metadata grouped in a single object saved in the cloud storage. This object is represented by a *Private Name Space* (PNS).

A PNS is a local data structure kept by the SCFS Agent's metadata service, containing the metadata of all private files of a user. Each PNS has an associated PNS tuple in the coordination service, which contains the user name and a reference to an object in the cloud storage. This object keeps a copy of the serialized metadata of all private files of the user.

Working with non-shared files is slightly different from what was shown in Figure 4. When mounting the file system, the agent fetches the user's PNS entry from the coordination service and the metadata from the cloud storage, locking the PNS to avoid inconsistencies caused by two clients logged in as the same user. When opening a file, the user gets the metadata locally as if it was in cache (since the file is not shared), and if needed fetches data from the cloud storage (as in the normal case). On close, if the file was modified, both data and metadata are updated in the cloud storage. The close operation completes when both updates finish.

When permissions change in a file, its metadata can be removed (resp. added) from a PNS, causing the creation (resp. removal) of the corresponding metadata tuple in the coordination service.

With PNSs, the amount of storage used in the coordination service is proportional to the percentage of shared files in the system. Previous work show traces with 1 million files where only 5% of them are shared [33]. Without PNSs, the metadata for these files would require 1 million tuples of around 1KB, for a total size of 1GB of storage (the approximate size of a metadata tuple is 1KB, assuming 100B file names). With PNSs, only 50 thousand tuples plus one PNS tuple per user would be needed, requiring a little more than 50MB of storage. Even more importantly, by resorting to PNSs, it is possible to reduce substantially the number of accesses to the coordination service, allowing more users and files to be served.

3 SCFS Implementation

SCFS is implemented in Linux as a user-space file system based on FUSE-J, which is a wrapper to connect the SCFS Agent to the FUSE library. Overall, the SCFS implementation comprises 6K lines of commented Java code, excluding any coordination service or storage backend code. We opted to develop SCFS in Java mainly because most of the backend code (the coordination and storage services) were written in Java and the high latency of cloud accesses make the overhead of using a Java-based file system comparatively negligible.

3.1 Modes of Operation

Our implementation of SCFS supports three modes of operation, based on the consistency and sharing requirements of the stored data.

The first mode, *blocking*, is the one described up to this point. The second mode, *non-blocking*, is a weaker version of SCFS in which closing a file does not block until the file data is on the clouds, but only until it is written locally and enqueued to be sent to the clouds in background. In this model, the file metadata is updated and the associated lock released only after the file contents are updated to the clouds, and not when the close call returns (so mutual exclusion is preserved). Naturally, this model leads to a significant performance improvement at cost of a reduction of the durability and consistency guarantees. Finally, the *non-sharing* mode is interesting for users that do not need to share files, and represents a design similar to S3QL [7], but with the possibility of using a cloud-of-clouds instead of a single storage service. This version does not require the use of the coordination service, and all metadata is saved on a PNS.

3.2 Backends

SCFS can be plugged to several backends, including different coordination and cloud storage services. This paper focuses on the two backends of Figure 5. The first one is based on Amazon Web Services (AWS), with an EC2 VM running the coordination service and file data being stored in S3. The second backend makes use of the cloud-of-clouds (CoC) technology, recently shown to be practical [9, 12, 15]. A distinct advantage of the CoC backend is that it removes any dependence of a single cloud provider, relying instead on a quorum of providers. It means that data security is ensured even if f out-of $3f+1$ of the cloud providers suffer *arbitrary faults*, which encompasses unavailability and data deletion, corruption or creation [15]. Although cloud providers have their means to ensure the dependability of their services, the recurring occurrence of outages, security incidents (with internal or external origins) and data corruptions [19, 24] justifies the need for this sort of backend in several scenarios.

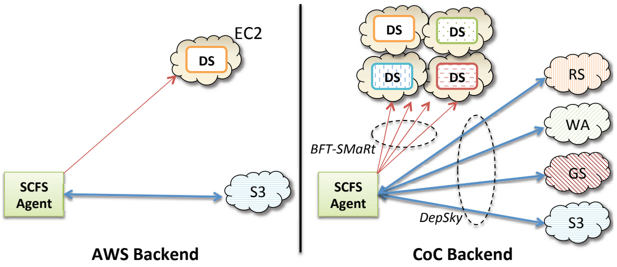


Figure 5: SCFS with Amazon Web Services (AWS) and Cloud-of-Clouds (CoC) backends.

Coordination services. The current SCFS prototype supports two coordination services: Zookeeper [29] and DepSpace [13] (in particular, its durable version [16]). These services are integrated at the SCFS Agent with simple wrappers, as both support storage of small data entries and can be used for locking. Moreover, these coordination services can be deployed in a replicated way for fault tolerance. Zookeeper requires $2f+1$ replicas to tolerate f crashes through the use of a Paxos-like protocol [30] while DepSpace uses either $3f+1$ replicas to tolerate f arbitrary/Byzantine faults or $2f+1$ to tolerate crashes (like Zookeeper), using the BFT-SMaRt replication engine [17]. Due to the lack of hierarchical data structures in DepSpace, we had to extend it with support for triggers to efficiently implement file system operations like rename.

Cloud storage services. SCFS currently supports Amazon S3, Windows Azure Blob, Google Cloud Storage, Rackspace Cloud Files and all of them forming a cloud-of-clouds backend. The implementation of single-cloud backends is simple: we employ the Java library made available by the providers, which accesses the cloud storage service using a REST API over SSL. To implement the cloud-of-clouds backend, we resort to an extended version of DepSky [15] that supports a new operation, which instead of reading the last version of a data unit, reads the version with a given hash, if available (to implement the consistency anchor algorithm - see §2.4). The hashes of all versions of the data are stored in DepSky's internal metadata object, stored in the clouds.

Figure 6 shows how a file is securely stored in the cloud-of-clouds backend of SCFS using DepSky (see [15] for details). The procedure works as follows: (1) a random key K is generated, (2) this key is used to encrypt the file and (3) the encrypted file is encoded and each block is stored in different clouds together with (4) a share of K, obtained through secret sharing. Stored data security (confidentiality, integrity and availability) is ensured by the fact that no single cloud alone has access to the data since K can only be recovered with two or more shares and that quorum reasoning is applied to discover the last version written. In the example of the figure, where a single faulty cloud is tolerated, two clouds need to be accessed to recover the file data.

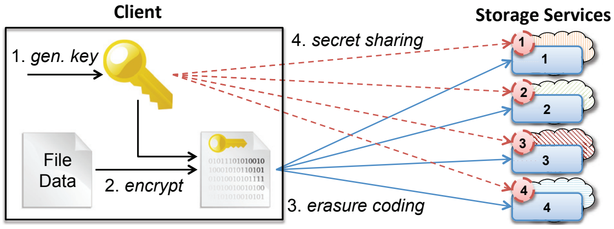


Figure 6: A write in SCFS using the DepSky protocols.

4 Evaluation

This section evaluates SCFS using AWS and CoC backends, operating in different modes, and comparing them with other cloud-backed file systems. The main objective is to understand how SCFS behaves with some representative workloads and to shed light on the costs of our design.

4.1 Setup & Methodology

Our setup considers a set of clients running on a cluster of Linux 2.6 machines with two quad-core 2.27 GHz Intel Xeon E5520, 32 GB of RAM and a 15K RPM SCSI HD. This cluster is located in Portugal.

For SCFS-AWS (Figure 5, left), we use Amazon S3 (US) as a cloud storage service and a single EC2 instance hosted in Ireland to run DepSpace. For SCFS-CoC, we use DepSky with four storage providers and run replicas of DepSpace in four computing cloud providers, tolerating a single fault both in the storage service and in the coordination service. The storage clouds were Amazon S3 (US), Google Cloud Storage (US), Rackspace Cloud Files (UK) and Windows Azure (UK). The computing clouds were EC2 (Ireland), Rackspace (UK), Windows Azure (Europe) and Elastichosts (UK). In all cases, the VM instances used were EC2 M1 Large [2] (or similar).

The evaluation is based on a set of benchmarks following recent recommendations [37], all of them from *Filebench* [3]. Moreover, we created two new benchmarks to simulate some behaviors of interest for cloud-backed file systems.

We compare six SCFS variants considering different modes of operation and backends (see Table 2) with two popular open source S3-backed files systems: S3QL [7] and S3FS [6]. Moreover, we use a FUSE-J-based local file system (LocalFS) implemented in Java as a baseline to ensure a fair comparison, since a native file system presents much better performance than a FUSE-J file system. In all SCFS variants, the metadata cache expiration time was set to 500 ms and no private name spaces were used. Alternative configurations are evaluated in §4.4.

	Blocking	*Non-blocking*	*Non-sharing*
AWS	SCFS-AWS-B	SCFS-AWS-NB	SCFS-AWS-NS
CoC	SCFS-CoC-B	SCFS-CoC-NB	SCFS-CoC-NS

Table 2: SCFS variants with different modes and backends.

4.2 Micro-benchmarks

We start with six Filebench micro-benchmarks [3]: sequential reads, sequential writes, random reads, random writes, create files and copy files. The first four benchmarks are IO-intensive and do not consider open, sync or close operations, while the last two are metadata-intensive. Table 3 shows the results for all considered file systems.

The results for sequential and random reads and writes show that the behavior of the evaluated file systems is similar, with the exception of S3FS and S3QL. The low performance of S3FS comes from its lack of main memory cache for opened files [6], while S3QL's low random write performance is the result of a known issue with FUSE that makes small chunk writes very slow [8]. This benchmark performs 4KB-writes, much smaller than the recommended chunk size for S3QL, 128KB.

The results for create and copy files show a difference of three to four orders of magnitude between the local or single-user cloud-backed file system (SCFS-*-NS, S3QL and LocalFS) and a shared or blocking cloud-backed file system (SCFS-*-NB, SCFS-*-B and S3FS). This is not surprising, given that SCFS-*-{NB,B} access the coordination service in each create, open or close operation. Similarly, S3FS accesses S3 in each of these operations, being even slower. Furthermore, the latencies of SCFS-*-NB variants are dominated by the coordination service access (between 60-100 ms per access), while in the SCFS-*-B variants such latency is dominated by read and write operations in the cloud storage.

4.3 Application-based Benchmarks

In this section we present two application-based benchmarks for potential uses of cloud-backed file systems.

File Synchronization Service. A representative workload for SCFS corresponds to its use as a personal file synchronization service [20] in which desktop application files (spreadsheets, documents, presentations, etc.) are stored and shared. A new benchmark was designed to simulate opening, saving and closing a text document with OpenOffice Writer.

Micro-benchmark	#Operations	File size	SCFS-AWS			SCFS-CoC			S3FS	S3QL	LocalFS
			NS	NB	B	NS	NB	B			
sequential read	1	4MB	1	1	1	1	1	1	6	1	1
sequential write	1	4MB	1	1	1	1	1	1	2	1	1
random 4KB-read	256k	4MB	11	11	15	11	11	11	15	11	11
random 4KB-write	256k	4MB	35	39	39	35	35	36	52	152	37
create files	200	16KB	1	102	229	1	95	321	596	1	1
copy files	100	16KB	1	137	196	1	94	478	444	1	1

Table 3: Latency of several Filebench micro-benchmarks for SCFS (six variants), S3QL, S3FS and LocalFS (in seconds).

The benchmark follows the behavior observed in traces of a real system, which are similar to other modern desktop applications [25]. Typically, the files managed by the cloud-backed file system are just copied to a temporary directory on the local file system where they are manipulated as described in [25]. Nonetheless, as can be seen in the benchmark definition (Figure 7), these actions (especially save) still impose a lot of work on the file system.

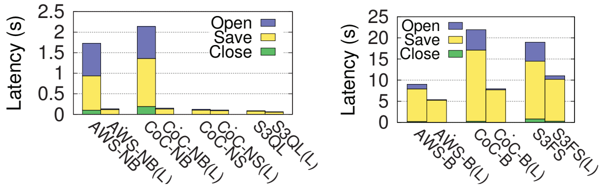
Open Action: 1 open(f,rw), 2 read(f), 3-5 open-write-close(lf1), 6-8 open-read-close(f), 9-11 open-read-close(lf1)

Save Action: 1-3 open-read-close(f), 4 close(f), 5-7 open-read-close(lf1), 8 delete(lf1), 9-11 open-write-close(lf2), 12-14 open-read-close(lf2), 15 truncate(f,0), 16-18 open-write-close(f), 19-21 open-fsync-close(f), 22-24 open-read-close(f), 25 open(f,rw)

Close Action: 1 close(f), 2-4 open-read-close(lf2), 5 delete(lf2)

Figure 7: File system operations invoked in the file synchronization benchmark, simulating an OpenOffice document open, save and close actions (*f* is the odt file and *lf* is a lock file).

Figure 8 shows the average latency of each of the three actions of our benchmark for SCFS, S3QL and S3FS, considering a file of 1.2MB, which corresponds to the average file size observed in 2004 (189KB) scaled-up 15% per year to reach the expected value for 2013 [11].



(a) Non-blocking.

(b) Blocking.

Figure 8: Latency of file synchronization benchmark actions (see Figure 7) with a file of 1.2MB. The (L) variants maintain lock files in the local file system. All labels starting with CoC or AWS represent SCFS variants.

Figure 8(a) shows that SCFS-CoC-NS and S3QL exhibit the best performance among the evaluated file systems, having latencies similar to a local file system (where a save takes around 100 ms). This shows that the added dependability of a cloud-of-clouds storage backend does not prevent a cloud-backed file system to behave similarly to a local file system, if the correct design is employed.

Our results show that SCFS-*-NB requires substantially more time for each phase due to the number of accesses to the coordination service, especially to deal with the lock files used in this workload. Nonetheless, saving a file in this system takes around 1.2 s, which is acceptable from the usability point of view. A much slower behavior is observed in the SCFS-*-B variants, where the creation of a lock file makes the system block waiting for this small file to be pushed to the clouds.

We observed that most of the latency comes from the manipulation of lock files. However, the files accessed did not need to be stored in the SCFS partition, since the locking service already prevents write-write conflicts between concurrent clients. We modified the benchmark to represent an application that writes lock files locally (in `/tmp`), just to avoid conflicts between applications in the same machine. The (L) variants in Figure 8 represent results with such local lock files. These results show that removing the lock files makes the cloud-backed system much more responsive. The takeaway here is that the usability of blocking cloud-backed file systems could be substantially improved if applications take into consideration the limitations of accessing remote services.

Sharing files. Personal cloud storage services are often used for sharing files in a controlled and convenient way [20]. We designed an experiment for comparing the time it takes for a shared file written by a client to be available for reading by another client, using SCFS-*-{NB,B}. We did the same experiment considering a Dropbox shared folder (creating random files to avoid deduplication). We acknowledge that the Dropbox design [20] is quite different from SCFS, but we think it is illustrative to show how a cloud-backed file system compares with a popular file synchronization service.

The experiment considers two clients A and B deployed in our cluster. We measured the elapsed time between the instant client A closes a variable-size file that it wrote to a shared folder and the instant it receives an UDP ACK from client B informing the file was available. Clients A and B are Java programs running in the same LAN, with a ping latency of around 0.2 ms, which is negligible considering the latencies of reading and writing. Figure 9 shows the results of this experiment for different file sizes.

The results show that the latency of sharing in SCFS-*-B is much smaller than what people experience in current personal storage services. These results do not consider

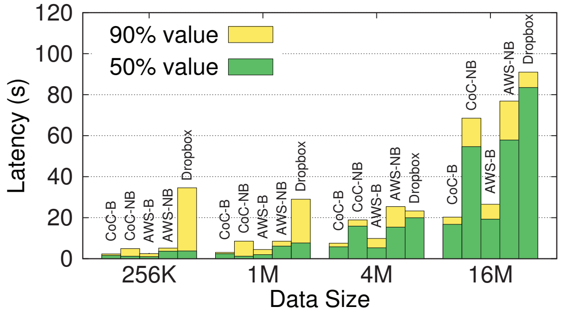


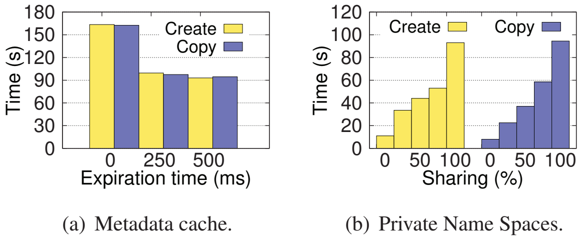
Figure 9: Sharing file 50th and 90th latency for SCFS (CoC B and NB, AWS B and NB) and Dropbox for different file sizes.

the benefits of deduplication, which SCFS currently does not support. However, if a user encrypts its critical files locally before storing them in Dropbox, the effectiveness of deduplication will be decreased significantly.

Figure 9 also shows that the latency of the blocking SCFS is much smaller than the non-blocking version with both AWS and CoC backends. This is explained by the fact that the SCFS-*-B waits for the file write to complete before returning to the application, making the benchmark measure only the delay of reading the file. This illustrates the benefits of SCFS-*-B: when A completes its file closing, it knows the data is available to any other client the file is shared with. We think this design can open interesting options for collaborative applications based on SCFS.

4.4 Varying SCFS Parameters

Figure 10 shows some results for two metadata-intensive micro-benchmarks (copy and create files) for SCFS-CoC-NB with different metadata cache expiration times and percentages of files in private name spaces.



(a) Metadata cache.

(b) Private Name Spaces.

Figure 10: Effect of metadata cache expiration time and PNSs with different file sharing percentages in two metadata intensive micro-benchmarks.

As described in §2.5.1, we implemented a short-lived metadata cache to deal with bursts of metadata access operations (e.g., `stat`). All previous experiments used an expiration time of 500 ms for this cache. Figure 10(a) shows how changing this value affects the performance of the system. The results clearly indicate that not using such metadata cache (expiration time equals zero) severely degrades the system performance. However, beyond some point, increasing it does not bring much benefit either.

Figure 10(b) displays the latency of the same benchmarks considering the use of PNS (see §2.7) with different percentages of files shared between more than one user. Recall that all previous results consider full-sharing (100%), without using PNS, which is a worst case scenario. As expected, the results show that as the number of private files increases, the performance of the system improves. For instance, when only 25% of the files are shared – more than what was observed in the most recent study we are aware of [33] – the latency of the benchmarks decreases by a factor of roughly 2.5 (create files) and 3.5 (copy files).

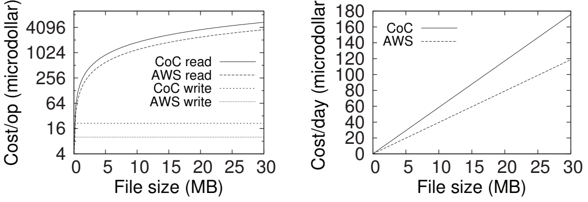
4.5 SCFS Operation Costs

Figure 11 shows the costs associated with operating and using SCFS. The *fixed operation costs* of SCFS comprise mainly the maintenance of the coordination service running in one or more VMs deployed in cloud providers. Figure 11(a) considers two instance sizes (as defined in Amazon EC2) and the price of renting one or four of them in AWS or in the CoC (one VM of similar size for each provider), together with the expected memory capacity (in number of 1KB metadata tuples) of such DepSpace setup. As can be seen in the figure, a setup with four Large instances would cost less than $1200 in the CoC per month while a similar setup in EC2 would cost $749. This difference of $451 can be seen as the operation cost of tolerating provider failures in our SCFS setup, and comes mainly from the fact that Rackspace and Elastichosts charge almost 100% more than EC2 and Azure for similar VM instances. Moreover, such costs can be factored among the users of the system, e.g., for one dollar per month, 2300 users can have a SCFS-CoC setup with Extra Large replicas for the coordination service. Finally, it is worth to mention that this fixed cost can be eliminated if the organization using SCFS hosts the coordination service in its own infrastructure.

Besides the fixed operation costs, each SCFS user has to pay for its usage (*executed operations* and *storage space*) of the file system. Figure 11(b) presents the cost of reading a file (open for read, read whole file and close) and writing a file (open for write, write the whole file, close) in SCFS-CoC and SCFS-AWS (S3FS and S3QL will have similar costs). The cost of reading a file is the only one that depends on the size of data, since providers charge around $0.12 per GB of outbound traffic, while inbound traffic is free. Besides that, there is also the cost of the `getMetadata` operation, used for cache validation, which is 11.32 microdollars ($\mu$$). This corresponds to the total cost of reading a cached file. The cost of writing is composed by metadata and lock service operations (see Figure 4), since inbound traffic is free. Notice that the design of SCFS exploits these two points: unmodified data is read locally and always written to the cloud for maximum durability.

VM Instance	EC2	EC2×4	CoC	Capacity
Large	$6.24	$24.96	$39.60	7M files
Extra Large	$12.96	$51.84	$77.04	15M files

(a) Operation costs/day and expected coordination service capacity.



(b) Cost per operation (log scale). (c) Cost per file per day.

Figure 11: The (fixed) operation and (variable) usage costs of SCFS. The costs include outbound traffic generated by the coordination service protocol for metadata tuples of 1KB.

Storage costs in SCFS are charged per number of files and versions stored in the system. Figure 11(c) shows the cost/version/day in SCFS-AWS and SCFS-CoC (considering the use of erasure codes and preferred quorums [15]). The storage costs of SCFS-CoC are roughly 50% more than of SCFS-AWS: two clouds store half of the file each while a third receives an extra block generated with the erasure code (the fourth cloud is not used).

It is also worth to mention that the cost of running the garbage collector corresponds to the cost of a list operation in each cloud ($\leq \mu$\$1/cloud), independently of the number of deleted files/versions. This happens because all used clouds do not charge delete operations.

5 Related Work

In this section we discuss some distributed file systems and cloud storage works that are most relevant to SCFS.

Cloud-backed file systems. S3FS [6] and S3QL [7] are two examples of cloud-backed file systems. Both these systems use unmodified cloud storage services (e.g., Amazon S3) as their backend storage. S3FS employs a blocking strategy in which every update on a file only returns when the file is written to the cloud, while S3QL writes the data locally and later pushes it to the cloud. An interesting design is implemented by BlueSky [39], another cloud-backed file system that can use cloud storage services as a storage backend. BlueSky provides a CIFS/NFS proxy (just as several commercially available cloud storage gateways) to aggregate writings in log segments that are pushed to the cloud in background, implementing thus a kind of log-structured cloud-backed file system. These systems differ from SCFS in many ways (see Figure 1), but mostly regarding their lack of controlled sharing support for geographically dispersed clients and dependency of a single cloud provider.

Some commercial cloud-enabled storage gateways [4, 5] also supports data sharing among proxies. These systems replicate file system metadata among the proxies, enabling one proxy to access files created by other proxies. Complex distributed locking protocols (executed by the proxies) are used to avoid write-write conflicts. In SCFS, a coordination service is used for metadata storage and lock management. Moreover, these systems neither support strongly consistent data sharing nor are capable to use a cloud-of-clouds backend.

Cloud-of-clouds storage. The use of multiple (unmodified) cloud storage services for data archival was first described in RACS [9]. The idea is to use RAID-like techniques to store encoded data in several providers to avoid vendor lock-in problems, something already done in the past, but requiring server code in the providers [31]. DepSky [15] integrates such techniques with secret sharing and Byzantine quorum protocols to implement single-writer registers tolerating arbitrary faults of storage providers. ICStore [12] showed it is also possible to build multi-writer registers with additional communication steps and tolerating only unavailability of providers. The main difference between these works and SCFS(-CoC) is the fact they provide a basic storage abstraction (a register), not a complete file system. Moreover, they provide strong consistency only if the underlying clouds provide it, while SCFS uses a consistency anchor (a coordination service) for providing strong consistency independently of the guarantees provided by the storage clouds.

Wide-area file systems. Starting with AFS [28], many file systems were designed for geographically dispersed locations. AFS introduced the idea of copying whole files from the servers to the local cache and making file updates visible only after the file is closed. SCFS adapts both these features for a cloud-backed scenario.

File systems like Oceanstore [32], Farsite [10] and WheelFS [36] use a small and fixed set of nodes as locking and metadata/index service (usually made consistent using Paxos-like protocols). Similarly, SCFS requires a small amount of computing nodes to run a coordination service and simple extensions would allow SCFS to use multiple coordination services, each one dealing with a subtree of the namespace (improving its scalability) [10]. Moreover, both Oceanstore [32] and Farsite [10] use PBFT [18] for implementing their metadata service, which makes SCFS-CoC superficially similar to their design: a limited number of nodes running a BFT state machine replication algorithm to support a metadata/coordination service and a large pool of untrusted storage nodes that archive data. However, on the contrary of these systems, SCFS requires few "explicit" servers, and only for coordination, since the storage nodes are replaced by cloud services like Amazon S3. Furthermore, these systems do not target controlled sharing of files and strong consistency, using thus long-term leases and weak cache coherence protocols. Finally, a distinctive feature of SCFS is that its design explicitly exploits the charging model of cloud providers.

6 Conclusions

SCFS is a cloud-backed file system that can be used for backup, disaster recovery and controlled file sharing, even without requiring trust on any single cloud provider. We built a prototype and evaluated it against other cloud-backed file systems and a file synchronization service, showing that, despite the costs of strong consistency, the design is practical and offers control of a set of tradeoffs related to security, consistency and cost-efficiency.

Acknowledgements. We thank the anonymous reviewers and Fernando Ramos for their comments to improve the paper. This work was supported by the EC's FP7 through projects BiobankCloud (317871) and TClouds (257243), and by the FCT through projects LaSIGE (PEst-OE/EEI/UI0408/2014) and INESC-ID (PEst-OE/EEI/LA0021/2013). Marcelo Pasin was funded by the EC's FP7 project LEADS (318809).

References

[1] 2012 future of cloud computing - 2nd annual survey results. http://goo.gl/fyrZFD.

[2] Amazon EC2 instance types. http://aws.amazon.com/ec2/instance-types/.

[3] Filebench webpage. http://sourceforge.net/apps/mediawiki/filebench/.

[4] Nasuni UniFS. http://www.nasuni.com/.

[5] Panzura CloudFS. http://panzura.com/.

[6] S3FS - FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/.

[7] S3QL - a full-featured file system for online data storage. http://code.google.com/p/s3ql/.

[8] S3QL 1.13.2 documentation: Known issues. http://www.rath.org/s3ql-docs/issues.html.

[9] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. *SoCC*, 2010.

[10] A. Adya et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[11] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study o file-system metadata. In *FAST*, 2007.

[12] C. Basescu et al. Robust data sharing with key-value stores. In *DSN*, 2012.

[13] A. Bessani, E. P. Alchieri, M. Correia, and J. S. Fraga. DepSpace: A Byzantine fault-tolerant coordination service. In *EuroSys*, 2008.

[14] A. Bessani and M. Correia. Consistency anchor formalization and correctness proofs. Technical Report DI-FCUL-2014-02, ULisboa, May 2014.

[15] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Trans. Storage*, 9(4), 2013.

[16] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the efficiency of durable state machine replication. In *USENIX ATC*, 2013.

[17] A. Bessani, J. Sousa, and E. Alchieri. State machine replication for the masses with BFT-SMaRt. In *DSN*, 2014.

[18] M. Castro and B. Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Trans. Computer Systems*, 20(4):398–461, 2002.

[19] S. Choney. Amazon Web Services outage takes down Netflix, other sites. http://goo.gl/t9pRbX, 2012.

[20] I. Drago et al. Inside Dropbox: Understanding personal cloud storage services. In *IMC*, 2012.

[21] G. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *ASPLOS*, 1998.

[22] A. Grünbacher. POSIX access control lists on Linux. In *USENIX ATC*, 2003.

[23] J. Hamilton. On designing and deploying Internet-scale services. In *LISA*, 2007.

[24] J. Hamilton. Observations on errors, corrections, and trust of dependent systems. http://goo.gl/LPTJoO, 2012.

[25] 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 *SOSP*, 2011.

[26] M. Herlihy. Wait-free synchronization. *ACM Trans. Programing Languages and Systems*, 13(1):124–149, 1991.

[27] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. on Programing Languages and Systems*, 12(3):463–492, 1990.

[28] J. Howard et al. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1):51–81, 1988.

[29] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale services. In *USENIX ATC*, 2010.

[30] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, 2011.

[31] R. Kotla, L. Alvisi, and M. Dahlin. SafeStore: A durable and practical storage system. In *USENIX ATC*, 2007.

[32] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[33] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC*, 2008.

[34] J. Li, M. N. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[35] Z. Li et al. Efficient batched synchronization in dropbox-like cloud storage services. In *Middleware*, 2013.

[36] J. Stribling et al. Flexible, wide-area storage for distributed system with WheelFS. In *NSDI*, 2009.

[37] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *HotOS*, 2011.

[38] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[39] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *FAST*, 2012.

[40] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.

[41] Y. Zhang, C. Dragga, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Viewbox: Integrating local file systems with cloud storage services. In *FAST*, 2014.

Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information

Min Fu[†], Dan Feng[†✉], Yu Hua[†], Xubin He[‡], Zuoning Chen[*], Wen Xia[†], Fangting Huang[†], Qing Liu[†]
[†]*Wuhan National Lab for Optoelectronics*
School of Computer, Huazhong University of Science and Technology, Wuhan, China
[‡]*Dept. of Electrical and Computer Engineering, Virginia Commonwealth University, VA, USA*
[*]*National Engineering Research Center for Parallel Computer, Beijing, China*
[✉]*Corresponding author: dfeng@hust.edu.cn*

Abstract

In deduplication-based backup systems, the chunks of each backup are physically scattered after deduplication, which causes a challenging fragmentation problem. The fragmentation decreases restore performance, and results in invalid chunks becoming physically scattered in different containers after users delete backups. Existing solutions attempt to rewrite duplicate but fragmented chunks to improve the restore performance, and reclaim invalid chunks by identifying and merging valid but fragmented chunks into new containers. However, they cannot accurately identify fragmented chunks due to their limited rewrite buffer. Moreover, the identification of valid chunks is cumbersome and the merging operation is the most time-consuming phase in garbage collection.

Our key observation that fragmented chunks remain fragmented in subsequent backups motivates us to propose a History-Aware Rewriting algorithm (HAR). HAR exploits historical information of backup systems to more accurately identify and rewrite fragmented chunks. Since the valid chunks are aggregated in compact containers by HAR, the merging operation is no longer required. To reduce the metadata overhead of the garbage collection, we further propose a Container-Marker Algorithm (CMA) to identify valid containers instead of valid chunks. Our extensive experimental results from real-world datasets show HAR significantly improves the restore performance by $2.6X$–$17X$ at a cost of only rewriting 0.45–1.99% data. CMA reduces the metadata overhead for the garbage collection by about $90X$.

1 Introduction

Deduplication has become a key component in modern backup systems due to its demonstrated ability of improving storage efficiency [26, 6]. A deduplication-based backup system divides a backup stream into variable-sized chunks [13], and identifies each chunk by its SHA-1 digest [19], i.e., *fingerprint*. A *fingerprint index* is used to map fingerprints of stored chunks to their physical addresses. In general, small and variable-sized chunks (e.g., 8KB on average [26]) are managed at a larger unit called *container* [26, 7, 9] that is a fixed-sized (e.g., 4MB [26]) structure. The containers are the basic unit of read and write operations. During a backup, the chunks that need to be written are aggregated into containers to preserve the locality of the backup stream. During a restore, a *recipe* (i.e., the fingerprint sequence of a backup) is read, and the containers serve as the prefetching unit. A restore cache holds the prefeteched containers and evicts an entire container via an LRU algorithm [9].

Since duplicate chunks are eliminated between multiple backups, the chunks of a backup unfortunately become physically scattered in different containers, which is known as fragmentation [18, 14]. First, the fragmentation severely decreases restore performance [15, 9]. The infrequent restore is important and the main concern from users [17]. Moreover, data replication, which is important for disaster recovery [20], requires reconstructions of original backup streams from deduplication systems [16], and thus suffers from a performance problem similar to the restore operation.

Second, the fragmentation results in invalid chunks (not referenced by any backups) becoming physically scattered in different containers when users delete expired backups. Existing solutions (i.e., reference management [7, 24, 4]) identify valid chunks and the containers holding only a few valid chunks. A merging operation is required to copy the valid chunks in the identified containers to new containers [10, 11], and then the identified containers are reclaimed. The merging is the most time-consuming phase in garbage collection [4].

A comprehensive category is helpful to understand the fragmentation. We observe that the fragmentation comes in two categories of containers: sparse containers and out-of-order containers. During a restore, a majority of chunks in a sparse container are never accessed, and the chunks in an out-of-order container are accessed intermittently. Both of them hurt the restore performance. Increasing the restore cache size alleviates the negative impacts of out-of-order containers, but it is ineffective for sparse containers because they directly amplify read operations (read many never accessed chunks). Additionally, the merging operation is required to reclaim sparse containers in the garbage collection after users delete backups.

Reducing sparse containers is important to address the fragmentation problem. Existing solutions [15, 8, 9] propose to rewrite duplicate but fragmented chunks during the backup via *rewriting algorithms*, which is a tradeoff between deduplication ratio (the size of the non-deduplicated data divided by that of the deduplicated data) and restore performance. These approaches buffer a small part of the backup stream, and identify the fragmented chunks within the buffer. They fail to identify sparse containers because an out-of-order container seems sparse in the limited-sized buffer. Hence, most of their rewritten chunks belong to out-of-order containers, which limit their gains in restore performance and garbage collection efficiency.

Our key observation is that two consecutive backups are very similar, and thus historical information collected during the backup is very useful to improve the next backup. For example, sparse containers for the current backup possibly remain sparse for the next backup. This observation motivates our work to propose a History-Aware Rewriting algorithm (HAR). During a backup, HAR rewrites the duplicate chunks in the sparse containers identified by the last backup, and records the emerging sparse containers to rewrite them in the next backup. HAR outperforms existing rewriting algorithms in terms of both restore performance and deduplication ratio. We also develop two optimization approaches for HAR to reduce the negative impacts of out-of-order containers on the restore performance, including an efficient restore caching scheme and a hybrid rewriting algorithm.

During the garbage collection, we need to identify valid chunks for identifying and merging sparse containers, which is cumbersome and error-prone due to the existence of large amounts of chunks. Since HAR efficiently reduces sparse containers, the identification of valid chunks is no longer necessary. We further propose a new reference management approach called Container-Marker Algorithm (CMA) that identifies valid containers (holding some valid chunks) instead of valid chunks. Comparing with existing reference management approaches, CMA significantly reduces the metadata overhead.

The paper makes the following contributions.

- We observe that the fragmentation is classified into two categories: out-of-order and sparse containers. The former reduces restore performance, which can be addressed by increasing the restore cache size. The latter reduces both restore performance and garbage collection efficiency, and we require a rewriting algorithm that is capable of accurately identifying sparse containers.

- In order to accurately identify and reduce sparse containers, we observe that sparse containers remain sparse in next backup, and hence propose HAR. HAR significantly improves restore performance with a slight decrease of deduplication ratio.

- In order to reduce the metadata overhead of the garbage collection, we propose CMA that identifies valid containers instead of valid chunks in the garbage collection.

The rest of the paper is organized as follow. Section 2 describes related work. Section 3 illustrates how the fragmentation arises. Section 4 discusses the fragmentation category and our observations. Section 5 presents our design and optimizations. Section 6 evaluates our approaches. Finally we conclude our work in Section 7.

2 Related Work

A deduplication system employs a large key-value subsystem, namely *fingerprint index*, to identify duplicates. The fingerprint index is too large to be completely stored in memory. However, a disk-based index that offers large-sized storage capacity suffers from severe performance bottleneck of accessing the fingerprints [19]. In order to address the performance problem of the fingerprint index, Zhu et al. [26] propose to leverage the locality of backup streams to accelerate fingerprint lookups. Extreme Binning [3], Sparse Index [10], and SiLo [25] mainly eliminate duplicate chunks among similar superchunks (consists of many chunks). ChunkStash [5] stores the index in SSDs instead of disks.

The fragmentation problem in deduplication systems has received many attentions. iDedup [21] eliminates sequential and duplicate chunks in the context of primary storage systems. Nam et al. propose a quantitative metric to measure the fragmentation level of deduplication systems [14], and a selective deduplication scheme [15] for backup workloads. The Context-Based Rewriting algorithm (CBR) [8] and the capping algorithm (CAP) [9] are recently proposed to address the fragmentation problem.

CBR uses a fixed-sized buffer, called *stream context*, to maintain the following chunks of the pending duplicate chunk that is being determined whether fragmented. CBR defines the *rewrite utility* of a pending chunk as the size of the chunks that are in the *disk context* (physically adjacent chunks) but not in the stream context, divided by the size of the disk context. If the rewrite utility of

Table 1: Existing reference management approaches.

Offline	Perfect Hash Vector [4]
Inline	Reference Counter [24], Grouped Mark-and-Sweep [7]

the pending chunk is higher than the predefined *minimal rewrite utility*, the chunk is fragmented. CBR uses a *rewrite limit* to avoid too many rewrites.

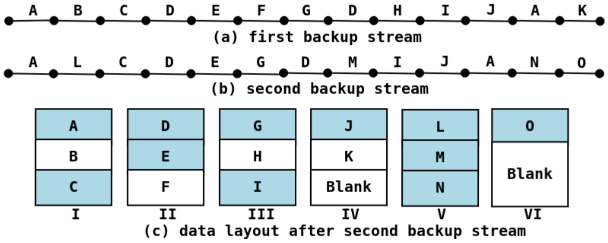
CAP divides the backup stream into fixed-sized segments, and conjectures the fragmentation within each segment. CAP limits the maximum number (say T) of containers a segment can refer to. Suppose a new segment refers to N containers and $N > T$, the chunks in the $N - T$ containers that hold the least chunks in the segment are rewritten.

Both of CBR and CAP buffer a small part of the ongoing backup stream during a backup, and identify fragmented chunks within the buffer (generally 10-20MB). They fail to accurately identify fragmented chunks, since physically adjacent chunks of a duplicate chunk can be accessed beyond the buffer. Increasing the buffer size alleviates this problem but is not scalable. Our approach is based on a new observation that fragmented chunks remain fragmented in the next backup, hence accurately identifying fragmented chunks.

Reference management for the garbage collection is complicated in deduplication systems, because each chunk can be referenced by multiple backups. Existing reference management approaches are summarized in Table 1. The offline approaches traverse all fingerprints (including the fingerprint index and recipes) when the system is idle. For example, Botelho et al. [4] build a perfect hash vector as a compact representation of all chunks. Since recipes need to occupy significantly large storage space [12], the traversing operation is time-consuming. The inline approaches maintain additional metadata during backup to facilitate the garbage collection. Maintaining a reference counter for each chunk [24] is expensive and error-prone [7]. Grouped Mark-and-Sweep (GMS) [7] uses a bitmap to mark which chunks in a container are used by a backup.

3 The Fragmentation Problem

Deduplication improves storage efficiency but causes fragmentation [18, 14], which exacerbates restore performance and garbage collection efficiency. Figure 1 illustrates an example of two consecutive backups to show how the fragmentation arises. There are 13 chunks in the first backup. Each chunk is identified by a character, and duplicate chunks share an identical character. Two duplicate chunks, say A and D, are identified by deduplicating the stream, which is called *self-reference*. A and D are called *self-referred chunks*. All unique chunks are stored in the first 4 containers, and a blank is appended to the $4th$ half-full container to make it be aligned. With



(a) first backup stream

(b) second backup stream

(c) data layout after second backup stream

Figure 1: An example of two consecutive backups. The shaded areas in each container represent the chunks required by the second backup.

a 3-container-sized LRU cache, restoring the first backup needs to read 5 containers. The self-referred chunk A requires extra reading container I.

We observe that the second backup contains 13 chunks, 9 of which are duplicates in the first backup. The four new chunks are stored in two new containers. With a 3-container-sized LRU cache, restoring the second backup needs to read 9 containers.

Although both of the backups consist of 13 chunks, restoring the second backup needs to read 4 more containers than restoring the first backup. Hence, the restore performance of the second backup is much worse than that of the first backup. Recent work [15, 8, 9] also reported the severe decrease of restore performance in deduplication systems. We observe a $21X$ decrease in our Linux dataset (detailed in Section 6.2).

If we delete the first backup, several chunks including chunk K in container IV become invalid. Because chunk J is still referenced by the second backup, we can't reclaim container IV. Existing work [10, 11] uses the offline container merging operation. The merging reads the containers that have only a few valid chunks and copies them to new containers. Therefore, it suffers from a performance problem similar to the restore operation, thus becoming the most time-consuming phase in the garbage collection [4].

4 Fragmentation Classification and Our Observations

We observe that the fragmentation comes in two categories: sparse containers and out-of-order containers. In this section, we describe these two types of containers and their impacts, and then present our key observations that motivate our work.

4.1 Sparse Container

As shown in Figure 1, only one chunk in container IV is referenced by the second backup. Prefetching container IV for chunk J is inefficient when restoring the second backup. After deleting the first backup, we require a merging operation to reclaim the invalid chunks in container IV. This kind of containers exacerbates system performance on both restore and garbage collection. We define a container's *utilization* for a backup as the fraction of its chunks referenced by the backup. If the utilization of a container is smaller than a predefined *utilization threshold*, such as 50%, the container is considered as a *sparse container* for the backup. We use the *average utilization* of all the containers related with a backup to measure the overall sparse level of the backup.

Sparse containers directly amplify read operations. Prefetching a container of 50% utilization at most achieves 50% of the maximum storage bandwidth, because 50% of the chunks in the container are never accessed. Hence, the average utilization determines the *maximum restore performance* with an unlimited restore cache. The chunks that have never been accessed in sparse containers require the slots in the restore cache, thus decreasing the available cache size. Therefore, reducing sparse containers can improve the restore performance.

After backup deletions, invalid chunks in a sparse container fail to be reclaimed until all other chunks in the container become invalid. Symantec [22] reports the probability that all chunks in a container become invalid is low. We also observe that garbage collection reclaims little space without additional mechanisms, such as offline merging sparse containers. Since the merging operation suffers from a performance problem similar to the restore operation, we require a more efficient solution to migrate valid chunks in sparse containers.

4.2 Out-of-order Container

If a container is accessed many times intermittently during a restore, we consider it as an *out-of-order container* for the restore. As shown in Figure 1, container V will be accessed 3 times intermittently while restoring the second backup. With a 3-container-sized LRU restore cache, restoring each chunk in container V incurs a cache miss that decreases restore performance.

The problem caused by out-of-order containers is complicated by self-references. The self-referred chunk *D* improves the restore performance, since the two accesses to *D* occur close in time. However, the self-referred chunk *A* decreases the restore performance.

The impacts of out-of-order containers on restore performance are related to the restore cache. For example, with a 4-container-sized LRU cache, restoring the three chunks in container V incurs only one cache miss. For each restore, there is a minimum cache size, called *cache threshold*, which is required to achieve the maximum restore performance (defined by the average utilization). Out-of-order containers reduce restore performance if the cache size is smaller than the cache threshold. They have no negative impact on garbage collection.

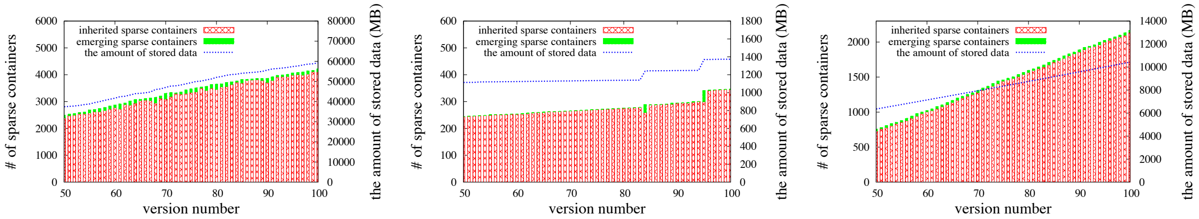
A sufficiently large cache can address the problem caused by out-of-order containers. However, since the memory is expensive, a restore cache of larger than the cache threshold can be unaffordable in practice. Hence, it is necessary to either decrease the cache threshold or assure the demanded restore performance if the cache is relatively small. If restoring a chunk in a container incurs an extra cache miss, it indicates that other chunks in the container are far from the chunk in the backup stream. Moving the chunk to a new container offers an opportunity to improve restore performance. Another more cost-effective solution to out-of-order containers is to develop a more intelligent caching scheme than LRU.

4.3 Our Observations

Because out-of-order containers can be alleviated by the restore cache, how to reduce sparse containers becomes the key problem. Existing rewriting algorithms cannot accurately identify sparse containers due to the limited buffer. Accurately identifying sparse containers requires the complete knowledge of the on-going backup. However, the complete knowledge of a backup cannot be known until the backup has concluded, making the identification of sparse containers a challenge.

Due to the incremental nature of backup, two consecutive backups are very similar, which is the major assumption behind DDFS [26]. Hence, they share similar characteristics, including the fragmentation. We analyze three datasets, including virtual machines, Linux kernels, and a synthetic dataset (detailed in Section 6.2), to explore and exploit potential characteristics of sparse containers (the utilization threshold is 50%). After each backup, we record the accumulative amount of the stored data, as well as the total and emerging sparse containers for the backup. An *emerging sparse container* is not sparse in the last backup but becomes sparse in the current backup. An *inherited sparse container* is already sparse in the last backup and remains sparse in the current backup. The total sparse containers are the sum of emerging and inherited sparse containers.

The characteristics of sparse containers are shown in Figure 2. First, the number of total sparse containers continuously grows. It indicates sparse containers become more common over time. Second, the number of total sparse containers increases smoothly most of time. A few exceptions in the Kernel datasets are major revision updates, which have more new data and increase the amount of stored data sharply. It indicates that a large update results in more emerging sparse containers. However, due to the similarity between consecutive backups, the number of emerging sparse containers of each backup is relatively small most of time. Third, the number of inherited sparse containers of each backup is equivalent to or slightly less than the number of total sparse containers of the previous backup. A few sparse containers of the previous backup become not sparse to the current backup since their utilizations drop to 0. It seldom occurs that the utilization of an inherited sparse container increases



(a) VMDK (b) Linux (c) Synthetic

Figure 2: Characteristics of sparse containers in three datasets. 50 backups are shown for clarity.

in the current backup, unless a rare rollback occurs. The observation indicates that sparse containers of the backup remain sparse in the next backup.

The above observations motivate our work to exploit the historical information to identify sparse containers. After completing a backup, we can determine which containers are sparse within the backup. Because these sparse containers remain sparse for the next backup, we record these sparse containers and allow chunks in them to be rewritten in the next backup. In such a scheme, the emerging sparse containers of a backup become the inherited sparse containers of the next backup. Due to the second observation, each backup needs to rewrite the chunks in a small number of inherited sparse containers, which would not degrade the backup performance. Moreover a small number of emerging sparse containers left to the next backup would not degrade the restore performance of the current backup. From the third observation, the scheme identifies sparse containers accurately. This scheme is called History-Aware Rewriting algorithm (HAR).

5 Design and Implementation

5.1 Architecture Overview

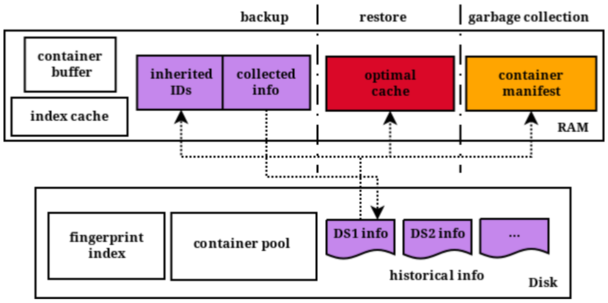


Figure 3: The HAR architecture.

Figure 3 illustrates the overall architecture of our HAR system. On disks, we have a container pool to provide container storage service. Any kinds of fingerprint indexes can be used. Typically we keep the complete fingerprint index on disks, as well as the hot part in memory. An in-memory container buffer is allocated for chunks to be written.

The system assigns each dataset a globally unique ID, such as $DS1$ in Figure 3. The collected historical information of each dataset is stored on disks with the dataset's ID, such as the $DS1$ $info$ file. The collected historical information consists of three parts: IDs of inherited sparse containers for HAR, the container-access sequence for the Belady's optimal replacement cache, and the container manifest for Container-Marker Algorithm.

5.2 History-Aware Rewriting Algorithm

At the beginning of a backup, HAR loads IDs of all inherited sparse containers to construct the in-memory $S_{inherited}$ structure, and rewrites all duplicate chunks in the inherited sparse containers. In practice, HAR maintains two in-memory structures, S_{sparse} and S_{dense} (included in *collected info* in Figure 3), to collect IDs of emerging sparse containers. The S_{sparse} traces the containers whose utilizations are smaller than the utilization threshold. The S_{dense} records the containers whose utilizations exceed the utilization threshold. The two structures consist of *utilization records*, and each record contains a container ID and the current utilization of the container. After the backup is completed, HAR replaces the IDs of the old inherited sparse containers with the IDs of emerging sparse containers in S_{sparse}. Hence, the S_{sparse} becomes the $S_{inherited}$ of the next backup. The complete workflow of HAR is described in Algorithm 1.

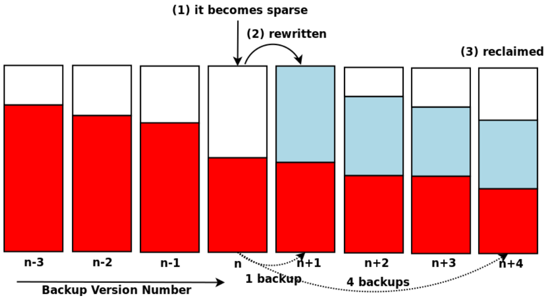


Figure 4: The lifespan of a rewritten sparse container.

Figure 4 illustrates the lifespan of a rewritten sparse container. The rectangle is a container, and the blank area is the chunks not referenced by the backup. We assume 4 backups are retained. (1) The container becomes sparse in backup n. (2) The container is rewritten in backup $n+1$. The chunks referenced by backup $n+1$ are rewritten to a new container that holds unique chunks and other

Algorithm 1 History-Aware Rewriting Algorithm

```
Input: IDs of inherited sparse containers, $S_{inherited}$;
Output: IDs of emerging sparse containers, $S_{sparse}$;
Initialize two sets, $S_{sparse}$ and $S_{dense}$.
while the backup is not completed do
    Receive a chunk and look up its fingerprint in the fingerprint index.
    if the chunk is duplicate then
        if the chunk's container ID exists in $S_{inherited}$ then
            Rewrite the chunk, and obtain a new container ID.
        else
            Eliminate the chunk.
        end if
    else
        Write the chunk, and obtain a new container ID.
    end if
    if the chunk's container ID doesn't exist in $S_{dense}$ then
        Update the associated utilization record (add it if doesn't exist) in $S_{sparse}$ with the chunk size.
        if the utilization exceeds the utilization threshold then
            Move the utilization record to $S_{dense}$.
        end if
    end if
end while
return $S_{sparse}$
```

rewritten chunks (blue area). However the old container cannot be reclaimed after backup $n+1$, because backup $n-2$, $n-1$, and n still refer to the old container. (3) After backup $n+4$ is finished, all backups referring to the old container have been deleted, and thus the old container can be reclaimed. Each sparse container decreases the restore performance of the backup recognizing it, and will be reclaimed when the backup is deleted.

Due to the limited number of inherited sparse containers, the memory consumed by the $S_{inherited}$ is negligible. S_{sparse} and S_{dense} consume more memory because they need to monitor all containers related with the backup. If the default container size is 4MB and the average utilization is 50% which can be easily achieved by HAR, the two sets of a 1TB stream consume 8MB memory (each record contains a 4-byte ID, a 4-byte current utilization, and an 8-byte pointer). This analysis shows that the memory footprint of HAR is low in most scenarios.

There is a tradeoff in HAR. A higher utilization threshold results in more containers being considered sparse, and thus backups are of better average utilization and restore performance but worse deduplication ratio. If the utilization threshold is set to 50%, HAR promises an average utilization of no less than 50%, and the maximum restore performance is no less than 50% of the maximum storage bandwidth.

5.2.1 The Impacts of HAR on Garbage Collection

We define C_i as the set of containers related with backup i, $|C_i|$ as the size of C_i, n_i as the number of inherited sparse containers, r_i as the size of rewritten chunks, and d_i as the size of new chunks. T backups are retained at any moment. The container size is S. The storage cost can be measured by the number of valid containers. A container is valid if it has chunks referenced by non-deleted backups. After backup k is finished, the number of valid containers is N_k.

$$N_k = |\bigcup_{i=k-T+1}^{k} C_i| = |C_{k-T+1}| + \sum_{i=k-T+2}^{k} (\frac{r_i + d_i}{S})$$

For those deleted backups (before backup $k-T+1$), we have

$$|C_{i+1}| = |C_i| - n_{i+1} + \frac{r_{i+1} + d_{i+1}}{S}, 0 \leq i < k-T+1$$

$$\Rightarrow N_k = |C_0| - \sum_{i=1}^{k-T+1} (n_i - \frac{r_i + d_i}{S}) + \sum_{i=k-T+2}^{k} (\frac{r_i + d_i}{S})$$

C_0 is the initial backup. Since the $|C_0|$, d_i, and S are constants, we concentrate on the part δ related with HAR,

$$\delta = - \sum_{i=1}^{k-T+1} (n_i - \frac{r_i}{S}) + \sum_{i=k-T+2}^{k} (\frac{r_i}{S}) \qquad (1)$$

The value of δ demonstrates the additional storage cost of HAR. If HAR is disabled (the utilization threshold is 0), δ is 0. A negative value of δ indicates that HAR decreases the storage cost. If k is small (the system is in the warn-up stage), the latter part is dominant thus HAR introduces additional storage cost than no rewriting. If k is large (the system is aged), the former part is dominant thus HAR decreases the storage cost.

A higher utilization threshold indicates that both n_i and r_i are larger. If k is small, a lower utilization threshold is helpful to decrease the storage cost since the latter part is dominant. Otherwise, the best utilization threshold is related with the backup retention time and characteristics of datasets. For example, if backups never expire, a higher utilization threshold always results in higher storage cost. Only retaining 1 backup would yield the opposite effect. However we find a value of 50% works well according to our experimental results in Section 6.7.

5.3 Optimal Restore Cache

To reduce the negative impacts of out-of-order containers on restore performance, we implement Belady's optimal replacement cache [2]. Implementing the optimal cache (OPT) needs to know the future access pattern. We can collect such information during the backup, since the sequence of reading chunks during the restore is just the same as the sequence of writing them during a backup.

After a chunk is processed through either elimination or over-writing its container ID, its container ID is known. We add an *access record* into the collected info in Figure 3. Each access record can only hold a container ID. Sequential accesses to the identical container can be merged into a record. This part of historical information can be updated to disks periodically, and thus would not consume much memory.

At the beginning of a restore, we load the container-access sequence into memory. If the cache is full, we evict the cached container that will not be accessed for the longest time in the future. Belady has proven the optimality [2].

The complete sequence of access records can consume considerable memory when out-of-order containers are dominant. Assuming each container is accessed 50 times intermittently and the average utilization is 50%, the complete sequence of access records of a 1TB stream consumes over 100MB of memory. Instead of checking the complete sequence of access records, we can use a slide window to check a fixed-sized part of the future sequence, as a near-optimal scheme. The memory footprint of this near-optimal scheme is hence bounded. Because the recent backups are most likely restored [8], we only maintain the sequences of a few recent backups for storage savings, and restore earlier backups via an LRU replacement caching scheme.

5.4 A Hybrid Scheme

As discussed in Section 4.2, rewriting chunks in out-of-order containers offers opportunities to reduce their negative impacts. Since most of the chunks rewritten by existing rewriting algorithms belong to out-of-order containers, we propose a hybrid scheme that takes advantages of both HAR and existing rewriting algorithms (e.g., CBR [8] and CAP [9]) as optional optimizations. The hybrid scheme is straightforward. Each duplicate chunk not rewritten by HAR is further examined by CBR or CAP. If CBR or CAP considers the chunk fragmented, the chunk is rewritten.

To avoid a significant decrease of deduplication ratio, we configure CBR or CAP to rewrite less data than the exclusive uses of themselves. For example, CBR uses a *rewrite limit* to control the rewrite ratio (the size of the rewritten chunks divided by that of the total chunks). The default rewrite limit in CBR is 5%, and thus CBR attempts to rewrite top-5% fragmented chunks. Generally a higher rewrite limit indicates CBR rewrites more data for higher restore performance. We set rewrite limit to 0.5% in the hybrid of HAR and CBR. The hybrid of HAR and CAP is similar. Based on our observations, only rewriting a small number of additional chunks further improves restore performance when the restore cache is small. However, the hybrid scheme always rewrites more data than HAR. Hence, we propose disabling the hybrid scheme if a large restore cache is affordable (Since restore is rare and critical, a large cache is reasonable).

5.5 Container-Marker Algorithm

Existing garbage collection schemes rely on merging sparse containers to reclaim invalid chunks in the containers. Before merging, they have to identify invalid chunks to determine utilizations of containers, i.e., reference management. Existing reference management approaches [24, 7, 4] are inevitably cumbersome due to the existence of large amounts of chunks.

HAR naturally accelerates expirations of sparse containers and thus the merging is no longer necessary. Hence, we need not to calculate the exact utilization of each container. We design the Container-Marker Algorithm (CMA) to efficiently determine which containers are invalid. CMA is fault-tolerant and recoverable.

CMA maintains a *container manifest* for each dataset. The container manifest records IDs of all containers related to the dataset. Each ID is paired with a backup time, and the backup time indicates the dataset's most recent backup that refers to the container. Each backup time can be represented by one byte, and let the backup time of the earliest non-deleted backup be 0. One byte suffices differentiating 256 backups, and more bytes can be allocated for longer backup retention time. Each container can be used by many different datasets. For each container, CMA maintains a dataset list that records IDs of the datasets referring to the container. A possible approach is to store the lists in the blank areas of containers, which on average is half of the chunk size. After a backup is completed, the backup time of the containers whose IDs are in the S_{sparse} and S_{dense} are updated to the largest time in the old manifest plus one. CMA adds the dataset's ID to the lists of the containers that are in the new manifest but not in the old one. If the lists (or manifests) are corrupted, we can recover them by traversing manifests of all datasets (or all related recipes).

If we need to delete the oldest t backups of a dataset, CMA loads the container manifest into memory. The container IDs with a backup time smaller than t are removed from the manifest, and the backup time of the remaining IDs decreases by t. CMA removes the dataset's ID from the lists of the removed containers. If a container's list is empty, the container can be reclaimed. We further examine the fingerprints in reclaimed containers. If a fingerprint is mapped to a reclaimed container in the fingerprint index, its entry is removed.

Because HAR effectively maintains high utilizations of containers, the container manifest is small. We as-

Table 2: Characteristics of datasets.

dataset name	VMDK	Linux	Synthetic
total size	1.44TB	104GB	4.5TB
# of versions	102	258	400
deduplication ratio	25.44	45.24	37.26
avg. chunk size	10.33KB	5.29KB	12.44KB
sparse	medium	severe	severe
out-of-order	severe	medium	medium

sume that each backup is 1TB and 90% identical to adjacent backups. Recent 20 backups are retained. With a 50% average utilization, the backups at most refer to 1.5 million containers. Hence the manifest and lists consume at most 13.5MB storage space (each container has a 4-byte container ID paired with a 1-byte backup time in the manifest, and a 4-byte dataset ID in its list).

6 Performance Evaluation

6.1 Experimental Configurations

We implemented an experimental platform to evaluate our design, including HAR, OPT, and CMA. We also implement CBR [8] (The original CBR is designed for HydraStor [6], and we implement the idea in the container storage), CAP [9], and their hybrid schemes (HAR+CBR and HAR+CAP) for comparisons. Since the design of fingerprint index is out of scope for the paper, we simply accommodate the complete fingerprint index in memory. The *baseline* has no rewriting, and the default caching scheme is OPT. The container size is 4MB. The default utilization threshold in HAR is 50%. We retain 20 backups thus backup $n-20$ is deleted after backup n is finished. We don't apply the offline container merging as in previous work [15, 9], because it requires a long idle time.

We use Speed Factor [9] as the metric of the restore performance. The speed factor is defined as 1 divided by mean containers read per MB of restored data. Higher speed factor indicates better restore performance. Given the container size is 4MB, 4 units of speed factor correspond to the maximum storage bandwidth.

6.2 Datasets

Two real-world datasets, including VMDK and Linux, and a synthetic dataset, i.e., Synthetic, are used for evaluation. Their characteristics are listed in Table 2. Each dataset is divided into variable-sized chunks.

VMDK is from a virtual machine installed Ubuntu 12.04LTS, which is a common use-case in real-world [7]. We compile source code, patch the system, and run an HTTP server on the virtual machine. We backup the virtual machine regularly. It consists of 102 full backups. Each full backup is 14.48GB on average, and 90–98% identical to its adjacent backups. Each backup contains about 15% self-referred chunks, and thus out-of-order containers are dominant.

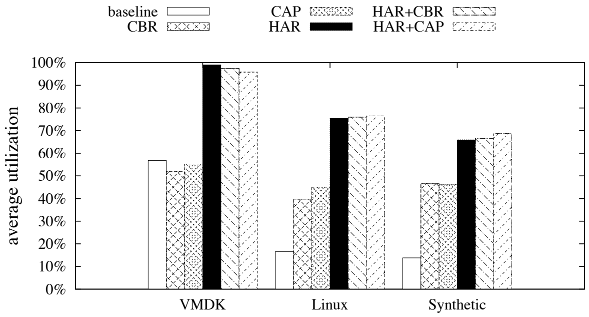


Figure 5: The average utilization of last 20 backups achieved by each rewriting algorithm.

Linux, downloaded from the web[1], is a commonly used public dataset [23]. It consists of 258 consecutive versions of unpacked Linux kernel sources. Each version is 412.78MB on average. Two consecutive versions are generally 99% identical except when there are large upgrades. In Linux, there are only a few self-references and sparse containers are dominant.

Synthetic is generated according to existing approaches [23, 9]. We simulate common operations of file systems, such as create/delete/modify files. We finally obtain a 4.5TB dataset with 400 versions. There is no self-reference in Synthetic.

6.3 Average Utilization

The average utilization of a backup exhibits its maximum restore performance. Figure 5 shows the average utilizations of rewriting algorithms. We observe that HAR significantly improves average utilizations, and obtains highest average utilizations in all datasets. The average utilizations of HAR are 99%, 75.42%, and 65.92% in VMDK, Linux, and Synthetic respectively, which indicate the *maximum speed factors* ($=$ *average utilization* $*$ 4) are 3.96, 3.02, and 2.64. CBR and CAP achieve lower average utilizations than the baseline in VMDK, because they rewrite many copies of self-referred chunks. They improve the average utilizations in Linux and Synthetic, although less than HAR by 30–50%. The hybrid schemes achieve average utilizations similar to HAR's.

6.4 Deduplication Ratio

Deduplication ratio explains the amount of written chunks, and the storage cost if no backup is deleted. Since we delete backups regularly to triggers garbage collection, the actual storage cost is shown in Section 6.6.

Figure 6 shows deduplication ratios of rewriting algorithms. The deduplication ratios of HAR are 22.78, 27.78, and 21.38 in VMDK, Linux, and Synthetic respectively. HAR rewrites 11.66%, 62.83%, and 74.31% more data than the baseline. However, the corresponding rewrite ratios remain at a low level, respectively 0.45%, 1.38%, and 1.99%. It indicates the size of rewritten

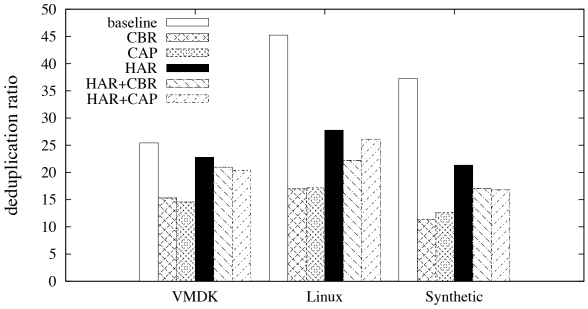


Figure 6: The comparisons between HAR and other rewriting algorithms in terms of deduplication ratio.

data is small relative to the size of backups. Due to such low rewrite ratios, the fingerprint lookup, content-defined chunking, and SHA-1 computation remain the performance bottleneck. Hence, HAR has trivial impacts on the backup performance.

We observe that HAR achieves considerably higher deduplication ratios than CBR and CAP. Since the rewrite ratios of CBR and CAP are 2 times larger than that of HAR, it is reasonable to expect that HAR outperforms CBR and CAP in terms of backup performance. The hybrid schemes, HAR+CBR and HAR+CAP, achieve better deduplication ratio than CBR and CAP respectively, but decrease deduplication ratios compared with HAR, such as by 10% in VMDK.

6.5 Restore Performance

Figure 7 shows the restore performance achieved by each rewriting algorithm with a given cache size. We tune the cache size according to the datasets, and show the impacts of varying cache size later in Figure 8. The default caching scheme is OPT. We observe severe declines of the restore performance in the baseline. For instance, restoring the latest backup is $21X$ slower than restoring the first backup in Linux. OPT alone increases restore performance by $1.51X$, $1.47X$, and $1.88X$ respectively in last 20 backups, however the performance remains at a low level.

We further examine the average speed factor in last 20 backups of each rewriting algorithm. In VMDK, CBR and CAP further improve restore performance by $1.46X$ and $1.53X$ respectively based on OPT. HAR outperforms them and increases restore performance by a factor of 1.72. The hybrid schemes are efficient, because HAR+CBR and HAR+CAP increase restore performance by $1.2X$ and $1.3X$ based on HAR. Given that their deduplication ratios are slightly smaller than HAR, CBR and CAP are good complements to HAR in the datasets where out-of-order containers are dominant. The restore performance of the initial backups exceeds the maximum storage bandwidth (4 units of speed factor), because self-referred chunks in the scope of the cache improve restore performance.

In Linux, CBR and CAP further improve restore performance by $5.4X$ and $6.12X$. HAR is more efficient and further increases restore performance by a factor of 10.25. Because out-of-order containers are less dominant, the hybrid schemes can't achieve significantly better performance than HAR. Thus the hybrid schemes can be disabled in the datasets where the problem of out-of-order containers is less severe. There are some occasional smaller values in the curve of HAR, because a large upgrade in Linux kernel produces a large amount of sparse containers.

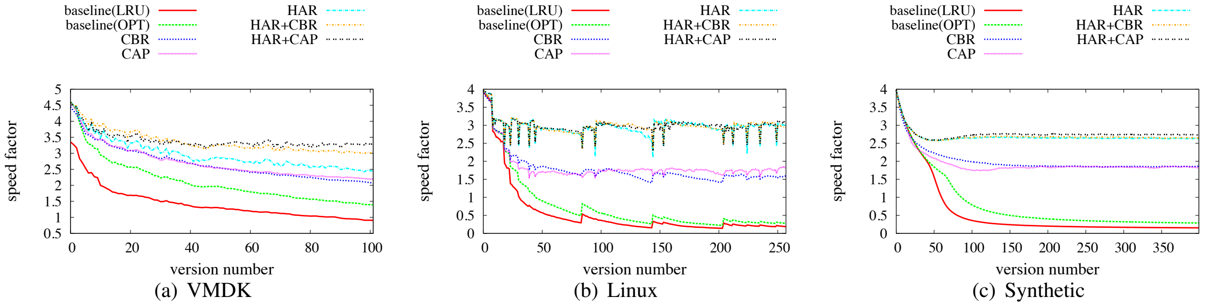
The results in Synthetic are similar with those in Linux. CBR, CAP, and HAR further increase restore performance by $6.41X$, $6.35X$, and $9.08X$ respectively. The hybrid schemes can't outperform HAR remarkably.

Figure 8 compares restore performance among rewriting algorithms under various cache sizes. In VMDK, because out-of-order containers are dominant, HAR requires a large cache (e.g., 2048-container-size) to achieve the maximum restore performance. We observe that if the cache size continuously increases, the restore performance of the baseline is approximate to that of CBR and CAP. The reason is that the baseline, CBR, and CAP achieve similar average utilizations as shown in Figure 5. CBR and CAP are great complements to HAR. When the cache is small, the restore performance of HAR+CBR (HAR+CAP) is approximate to that of CBR (CAP); when the cache is large, the restore performance of the hybrid schemes is approximate to that of HAR. Compared with HAR, the hybrid schemes successfully decrease the cache threshold by nearly $2X$, and improve the restore performance when the cache is small.

In Linux, HAR achieves better restore performance than CBR and CAP, even with a small cache (e.g., 8-container-size). Compared with HAR, the hybrid schemes decrease the cache threshold by a factor of 2, and improve the restore performance when the cache is small. However, because the cache threshold of HAR is small, a restore cache of reasonable size can address the problem caused by out-of-order containers without decreasing deduplication ratio.

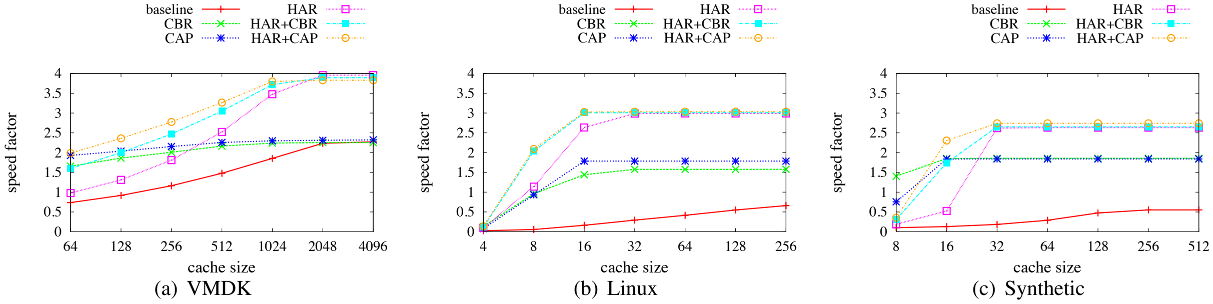
In Synthetic, HAR outperforms CBR and CAP by $1.41X$ and $1.42X$ when the cache is no less than 32-container-size. With a small cache (e.g., 8-container-size), CBR and CAP are better. However, because the cache threshold of HAR is small, it is reasonable to allocate sufficient memory for a restore. The hybrid schemes improve restore performance when the cache is small.

The experimental maximum restore performance in each dataset verifies our estimated values in Section 6.3. In summary, we propose to use the hybrid schemes when self-references are common; otherwise the exclusive use of HAR is recommended.



(a) VMDK (b) Linux (c) Synthetic

Figure 7: The comparisons of rewriting algorithms in terms of restore performance. The cache is 512-, 32-, and 64-container-sized in VMDK, Linux, and Synthetic respectively.



(a) VMDK (b) Linux (c) Synthetic

Figure 8: The comparisons of rewriting algorithms under various cache size. Speed factor is the average value of last 20 backups. The cache size is in terms of # of containers.

Table 3: Metadata space overhead of inline reference management approaches. HAR is used in all approaches.

	VMDK	**Linux**	**Synthetic**
Reference Counter [24]	4.64MB	328.36KB	6.53MB
GMS [7]	5.26MB	190KB	7.23MB
CMA	58.19KB	2KB	81.62KB

6.6 Garbage Collection

We compare the metadata space overhead among existing inline reference management approaches in Table 3. We assume each reference counter consumes one byte. The metadata overhead of CMA is lowest, and no more than 1/90 of that of GMS.

We examine how rewriting algorithms affect garbage collection. The number of valid containers after garbage collection exhibits the actual storage cost, and the results are shown in Figure 9. In the initial backups, the baseline has least valid containers, which verifies the discussions in Section 5.2.1. The advantage of HAR becomes more apparent over time, since the proportion of the former part in Equation 1 increases. Finally HAR decreases the number of valid containers by 27.37%, 68.15%, and 68.43% compared to the baseline in VMDK, Linux, and Synthetic respectively. In Synthetic, the number of valid containers increases continuously because the data size increases. The results indicate HAR achieves better storage saving than the baseline, and the merging is no longer necessary in a deduplication system with HAR.

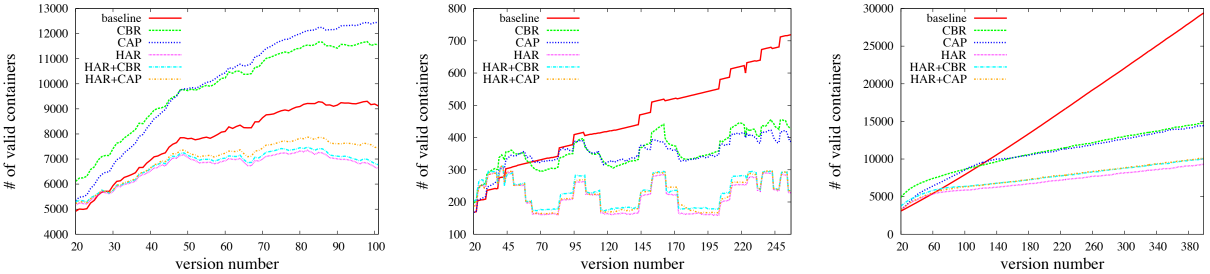
We observe that CBR and CAP increase the number of valid containers by 26.8% and 36.47% respectively in VMDK compared to the baseline. It indicates that CBR and CAP exacerbate the problem of garbage collection in VMDK. The reason is that they rewrite many copies of self-referred chunks into different containers, which reduces the average utilizations as shown in Figure 5. In Linux and Synthetic, CBR and CAP reduce the number of valid containers by 50%, however they still require the merging operation to achieve further storage savings.

HAR+CBR and HAR+CAP respectively result in 2.3% and 12.5% more valid containers than HAR in VMDK. However they significantly reduce the number of valid containers compared with the baseline. They perform slightly worse than HAR in Linux and Synthetic, and outperform CBR and CAP in all three datasets.

6.7 Varying the Utilization Threshold

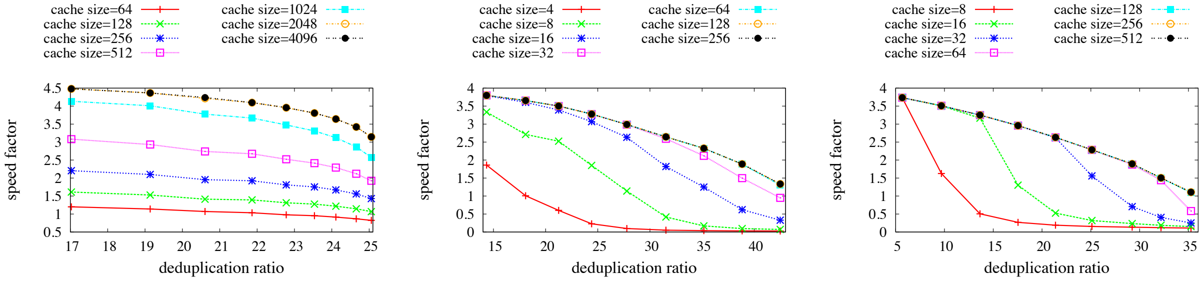
The utilization threshold determines the definition of sparse containers. The impacts of varying the utilization threshold on deduplication ratio and restore performance are both shown in Figure 10.

Varying the utilization threshold from 90% to 10%, the deduplication ratio increases from 17.03 to 25.06 and the restore performance decreases by about 35% in VMDK. In particular, with a 70% utilization threshold and a 2048-container-sized cache, the restore performance exceeds 4 units of speed factor. The reason is that the self-referred chunks restore more data than them-



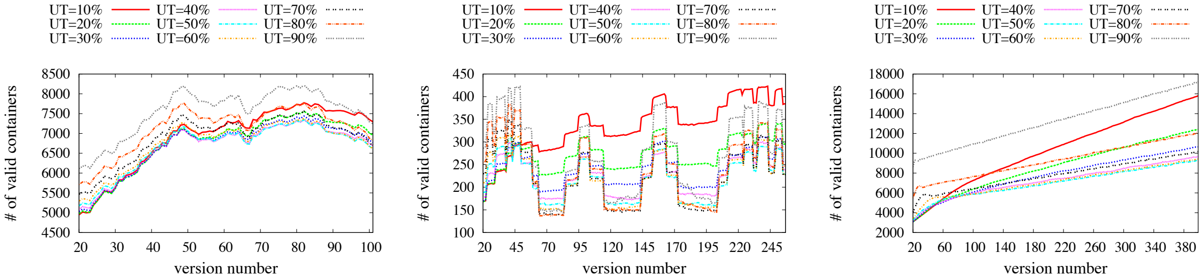
(a) VMDK (b) Linux (c) Synthetic

Figure 9: The comparisons of rewriting algorithms in terms of garbage collection.



(a) VMDK (b) Linux (c) Synthetic

Figure 10: Impacts of varying the utilization threshold on restore performance and deduplication ratio. Speed factor is the average value of last 20 backups. The cache size is in terms of # of containers. Each curve shows varying the utilization threshold from left to right: 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, and 10%.



(a) VMDK (b) Linux (c) Synthetic

Figure 11: Impacts of varying the Utilization Threshold (UT) on garbage collection.

selves. In Linux and Synthetic, deduplication ratio and restore performance are more sensitive to the change of the utilization threshold than in VMDK. Varying the utilization threshold from 90% to 10%, the deduplication ratio increases from 14.34 to 42.49, and 5.68 to 35.26 respectively. The smaller the restore cache is, the more significant the performance decrease is as the utilization threshold decreases.

Varying the utilization threshold also has significant impacts on garbage collection. The results are shown in Figure 11. A lower utilization threshold results in less valid containers in initial backups of all our datasets. However, we observe a trend that higher utilization thresholds gradually outperform lower utilization thresholds over time. For instance, the best utilization threshold finally is 50–60% in VMDK, 50–70% in Linux, and 50% in Synthetic. There are some periodical peaks in Linux, since a large upgrade to kernel results in a large amount of emerging sparse containers. These containers will be rewritten in the next backup, which suddenly increases the number of valid containers. After the backup expires, the number of valid containers is reduced.

Based on the experimental results, we believe a 50% threshold is practical in most cases, since it causes moderate rewrites and obtains significant improvements in restore and garbage collection.

7 Conclusions

The fragmentation decreases the efficiencies of restore and garbage collection in deduplication-based backup systems. We observe that the fragmentation comes in two categories: sparse containers and out-of-order containers. Sparse containers determine the maximum restore performance of a backup while out-of-order containers determine the required cache size to achieve the maximum restore performance.

History-Aware Rewriting algorithm (HAR) accurately identifies and rewrites sparse containers via exploiting historical information. We also implement an optimal restore caching scheme (OPT) and propose a hybrid rewriting algorithm as complements of HAR to reduce the negative impacts of out-of-order containers. HAR, as well as OPT, improves restore performance by $2.6X$–$17X$ at an acceptable cost in deduplication ratio. HAR outperforms the state-of-the-art work in terms of both deduplication ratio and restore performance. The hybrid schemes are helpful to further improve restore performance in datasets where out-of-order containers are dominant.

The ability of HAR to reduce sparse containers facilitates the garbage collection. It is no longer necessary to offline merge sparse containers, which relies on identifying valid chunks. We propose a Container-Marker Algorithm (CMA) that identifies valid containers instead of valid chunks. Since the metadata overhead of CMA is bounded by the number of containers, it is more cost-effective than existing reference management approaches whose overhead is bounded by the number of chunks.

Acknowledgments

The work was partly supported by National Basic Research 973 Program of China under Grant No. 2011CB302301; NSFC No. 61025008, 61173043, and 61232004; 863 Project 2013AA013203; Electronic Development fund of Information Industry Ministry. The work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China. The work conducted at VCU was partly supported by US National Science Foundation (NSF) Grants CCF-1102624 and CNS-1218960. The authors are also grateful to Jon Howell and anonymous reviews for their feedback.

References

[1] Linux kernel. http://www.kernel.org/, 2013.

[2] BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM systems journal 5*, 2 (1966), 78–101.

[3] BHAGWAT, D., ESHGHI, K., LONG, D. D., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. IEEE MASCTOS, 2009*.

[4] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *Proc. USENIX FAST, 2013*.

[5] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proc. USENIX FAST, 2010*.

[6] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. HYDRAstor: A scalable secondary storage. In *Proc. USENIX FAST, 2009*.

[7] GUO, F., AND EFSTATHOPOULOS, P. Building a highperformance deduplication system. In *Proc. USENIX ATC, 2011*.

[8] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proc. ACM SYSTOR, 2012*.

[9] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. USENIX FAST, 2013*.

[10] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. USENIX FAST, 2009*.

[11] MEISTER, D., AND BRINKMANN, A. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proc. IEEE MSST, 2010*.

[12] MEISTER, D., BRINKMANN, A., AND SÜSS, T. File recipe compression in data deduplication systems. In *Proc. USENIX FAST, 2013*.

[13] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proc. ACM SOSP, 2001*.

[14] NAM, Y., LU, G., PARK, N., XIAO, W., AND DU, D. H. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proc. IEEE HPCC, 2011*.

[15] NAM, Y. J., PARK, D., AND DU, D. H. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proc. IEEE MASCOTS, 2012*.

[16] POSEY, B. Deduplication and data lifecycle management. http://searchdatabackup.techtarget.com/tip/Deduplication-and-data-lifecycle-management, 2013.

[17] PRESTON, W. C. *Backup & Recovery*. O'Reilly Media, Inc., 2006.

[18] PRESTON, W. C. Restoring deduped data in deduplication systems. http://searchdatabackup.techtarget.com/feature/Restoring-deduped-data-in-deduplication-systems, 2010.

[19] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proc. USENIX FAST, 2002*.

[20] SHILANE, P., HUANG, M., WALLACE, G., AND HSU, W. WAN-optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (TOS) 8*, 4 (2012), 13.

[21] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. USENIX FAST, 2012*.

[22] SYMANTEC. How to force a garbage collection of the deduplication folder. http://www.symantec.com/business/support /index?page=content&id=TECH129151, 2010.

[23] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *Proc. USENIX ATC, 2012*.

[24] WEI, J., JIANG, H., ZHOU, K., AND FENG, D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proc. IEEE MSST, 2010*.

[25] 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 *Proc. USENIX ATC, 2011*.

[26] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. USENIX FAST, 2008*.

The TURBO Diaries: Application-controlled Frequency Scaling Explained

Jons-Tobias Wamhoff
Stephan Diestelhorst
Christof Fetzer
Technische Universtät Dresden, Germany

Patrick Marlier
Pascal Felber
Université de Neuchâtel, Switzerland

Dave Dice
Oracle Labs, USA

Abstract

Most multi-core architectures nowadays support dynamic voltage and frequency scaling (DVFS) to adapt their speed to the system's load and save energy. Some recent architectures additionally allow cores to operate at boosted speeds exceeding the nominal base frequency but within their thermal design power.

In this paper, we propose a general-purpose library that allows selective control of DVFS from user space to accelerate multi-threaded applications and expose the potential of heterogeneous frequencies. We analyze the performance and energy trade-offs using different DVFS configuration strategies on several benchmarks and real-world workloads. With the focus on performance, we compare the latency of traditional strategies that halt or busy-wait on contended locks and show the power implications of boosting of the lock owner. We propose new strategies that assign heterogeneous and possibly boosted frequencies while all cores remain fully operational. This allows us to leverage performance gains at the application level while all threads continuously execute at different speeds. We also derive a model to help developers decide on the optimal DVFS configuration strategy, e.g, for lock implementations. Our in-depth analysis and experimental evaluation of current hardware provides insightful guidelines for the design of future hardware power management and its operating system interface.

1 Introduction

While early generations of multi-core processors were essentially homogeneous with all cores operating at the same clock speed, new generations provide finer control over the frequency and voltage of the individual cores. A major motivation for this new functionality is to maximize processor performance without exceeding the thermal design power (TDP), as well as reducing energy consumption by decelerating idle cores [4, 35].

Two main CPU manufacturers, Intel and AMD, have proposed competing yet largely similar technologies for dynamic voltage and frequency scaling (DVFS) that can exceed the processor's nominal operation frequency, respectively named *Turbo Boost* [39] and *Turbo CORE* [3]. When the majority of cores are powered down or run at a low frequency, the remaining cores can boost within the limits of the TDP. In the context of multi-threaded applications, a typical use case is the optimization of sequential bottlenecks: waiting threads halt the underlying core and allow the owner thread to speed up execution of the critical section.

Boosting is typically controlled by hardware and is completely transparent to the operating system (OS) and applications. Yet, it is sometimes desirable to be able to finely control these features from an application as needed. Examples include: accelerating the execution of key sections of code on the critical path of multi-threaded applications [9]; boosting time-critical operations or high-priority threads; or reducing the energy consumption of applications executing low-priority threads. Furthermore, workloads specifically designed to run on processors with heterogeneous cores (e.g., few fast and many slow cores) may take additional advantage of application-level frequency scaling. We argue that, in all these cases, fine-grained tuning of core speeds requires *application knowledge* and hence cannot be efficiently performed by hardware only.

Both Intel and AMD hardware implementations are constrained in several ways, e.g., some combination of frequencies are disallowed, cores must be scaled up/down in groups, or the CPU hardware might not comply with the scaling request in some circumstances. Despite the differences of both technologies, our comparative analysis derives a common abstraction for the processor performance states (Section 2). Based on the observed properties, we present the design and implementation of TURBO, a general-purpose library for application-level DVFS control that can programmatically configure the speed of the cores of CPUs with AMD's Turbo CORE and Intel's Turbo Boost technologies, while abstracting the low-level differences and complexities (Section 3).

The cost of frequency and voltage transitions is subject to important variations depending on the method used for modifying processor states and the specific change requested. The publicly available documentation is sparse, and we believe to be the first to publish an in-depth investigation on the latency, performance, and limitations of these DVFS technologies (Section 4). Unlike previous research, our goal is not energy conservation or thermal boosting [36], which is usually applied to mobile devices and interactive applications with long idle periods, but long running applications often found on servers. We target efficiency by focusing on the best performance, i.e., shorter run times or higher throughput using the available TDP. In this context, hardware is tuned in combination with the OS to use frequency scaling for boosting sequential bottlenecks on the critical path of multi-threaded applications. We use the TURBO library to measure the performance and power implications of both blocking and spinning locks (Section 4.2). Our evaluation shows that connecting knowledge of application behavior to programmatic control of DVFS confers great benefits on applications having heterogeneous load. We propose new configuration strategies that keep all cores operational and allow a manual boosting control (Section 4.3).

Based on the evaluation of manual configuration strategies

	AMD FX-8120	**Intel i7-4770**
Model	AMD Family 15h Model 1	Intel Core 4th generation
Codename	“Bulldozer”	“Haswell”
Design	4 modules with 2 ALUs & 1 FPU	4 cores with hyper-threading
L2 cache	4×2MB per module	4×256KB per core
L3 cache	1×8MB per package	1×8MB per package
TDP	124.95W (NB 14.23W)	84W
Frequency	3.1GHz, (1.4–4.0GHz)	3.4GHz (0.8–3.9GHz)
Stepping	ACPI P-states, 100MHz	multiplier P-states, 100MHz
Voltage	0.875–1.412V (3.41–27.68W)	0.707–1.86V (5–75W)

Table 1: Specification of the AMD and Intel processors.

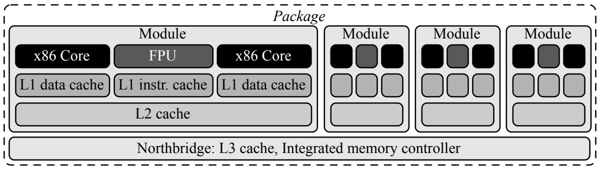


Figure 1: Organization of an AMD FX-8120 processor.

and their latencies, we derive a simplified cost model (Section 4.4) to guide developers at which size of a critical region a frequency transition pays off. Four case studies investigate the performance gains exploited by application-level frequency control based on real-world benchmarks (Section 5).

2 Hardware Support for Boosting

With both AMD’s Turbo CORE and Intel’s Turbo Boost, performance levels and power consumption of the processor are controlled through two types of operational states: *P-states* implement DVFS and set different frequency/voltage pairs for operation, trading off higher voltage (and thus higher power draw) with higher performance through increased operation frequency. P-states can be controlled through special machine-specific registers (MSRs) that are accessed through the `rdmsr/wrmsr` instructions. The OS can request a P-state change by modifying the respective MSR. P-state changes are also not instantaneous: the current needs to be adapted and frequencies are ramped, both taking observable time.

C-states are used to save energy when a core is idle. C0 is the normal operational state. All other C-states halt the execution of instructions and trade different levels of entry/wakeup latency for lower power draw. The OS can invoke C-states through various means such as the `hlt` and `monitor/mwait` instructions. We argue in this paper that there are benefits in keeping selected cores operational, albeit at a lower frequency, and that manipulating P-states can be more efficient in terms of latency than manipulating C-states.

We base our work on AMD’s FX-8120 [1] and Intel’s i7-4770 [19] CPUs, whose characteristics are listed in Table 1.

2.1 AMD’s Turbo CORE

The architecture of the AMD FX-8120 processor is illustrated in Figure 1. The *cores* of a *package* are organized by pairs in *modules* that share parts of the logic between the two cores.

Our processor supports seven P-states summarized in Table 2. We introduce a TURBO naming convention to abstract from the manufacturer specifics. AMD uses P-state numbering based on the ACPI standard with P0 being the highest performance state. The two topmost are boosted P-states (#$P_{boosted}$ $= 2$) that are by default controlled by the hardware. The remaining five P-states can be set by the OS through the MSRs[1].

Hardware P-state	P0	P1	**P2**	P3	P4	P5	P6
TURBO naming	P_{turbo}		P_{base}				P_{slow}
Frequency (GHz)	4.0	3.4	3.1	2.8	2.3	1.9	1.4
Voltage (mV)	1412	1412	1275	1212	1087	950	875
Power 4×`nop` (W)	—	123.3	113.6	97.2	70.1	49.9	39.3
Power 4×*ALU* (W)	—	—	122.6	104.3	74.6	52.9	41.2
Power 3×P_{slow}, 1×P0..6 (W)	125.0	119.8	100.5	87.4	65.5	48.5	41.2
Power 3×`mwait`, 1×P0..6 (W)	120.1	116.5	90.9	77.6	55.5	40.5	32.8

Table 2: Default P-state configuration of AMD FX-8120.

The boosting of the frequency beyond the nominal P-state (P_{base}) is enabled by the hardware’s Turbo CORE technology if operating conditions permit. The processor determines the current power consumption and will enable the first level of boosting ($P1_{HW}$) if the total power draw remains within the TDP limit and the OS requests the fastest software P-state. A multi-threaded application can boost one module to $P1_{HW}$ while others are in P_{base} if it does not use all features of the package to provide the required power headroom, e.g., no FPUs are active. The fastest boosting level (P_{turbo}) is entered automatically if some cores have furthermore reduced their power consumption by entering a deep C-state. Note that Turbo CORE is deterministic, governed only by power draw and not temperature, such that the maximum frequency is workload dependent. During a P-state transition, the processor remains active and capable of executing instructions, and the completion of a P-state transition is indicated in an MSR available to the OS.

The Turbo CORE features can be enabled or disabled altogether, i.e., no core will run above P_{base}. Selected AMD processors allow developers to control the number of hardware-reserved P-states by changing #$P_{boosted}$ through a configuration MSR. To achieve manual control over all P-states, including boosting, one can set #$P_{boosted} = 0$. The core safety mechanisms are still in effect: the hardware only enters a boosted P-state if the TDP limit has not been reached. In contrast to the processor’s automatic policy, the manual control of all P-states can enable P_{turbo} with all other cores in C0 but running at P_{slow}.

Due to the pairwise organization of cores in modules, the effect of a P- and C-state change depends on the state of the sibling core. While neighboring cores can request P-states independently, the fastest selected P-state of the two cores will apply to the entire module. Since the `wrmsr` instruction can only access MSRs of the current core, it can gain full control over the frequency scaling if the other core is running at P_{slow}. A module only halts if both cores are not in C0.

The processor allows to read the current power draw (P) that it calculates based on the load. Out of the total TDP, 14.24W are reserved for the northbridge (NB) (including L3 cache) and logic external to the cores. Each of the four modules is a voltage (V) and frequency (f) domain defined by the P-state. The package requests V defined by the fastest active P-state of any module from the voltage regulator module (VRM).

[1] The numbering in software differs from the actual hardware P-states: $P_{HW} = P_{SW} + \#P_{boosted}$. With a default of $\#P_{boosted} = 2$: $P_{base} = P0_{SW} = P2_{HW}$ and $P_{turbo} = P0_{HW}$. $P0_{SW}$ is the fastest requestable software P-state.

Hardware P-state	P39	**P34**	P20	P8
TURBO naming	P_{turbo}	P_{base}		P_{slow}
Frequency (GHz)	3.9	3.4	2.0	0.8
Voltage (mV)	1860	n/a	n/a	707
Power `nop` (W)	—	39	20	11
Power *ALU* (W)	—	51	25	12
Power `mwait` (W)	25	19	11	8

Table 3: Default P-state configuration of Intel i7-4770.

Table 2 lists P with (1) all cores in the same P-state executing `nop` instructions, (2) execution of integer operations with *ALU*, (3) three modules in P_{slow} except one in the given P-state, and (4) all modules halted using `mwait` except one active core. The consumed active P depends on V, f and the capacitance (C) that varies dynamically with the workload ($P = V^2 * f * C_{dyn}$). Therefore, for the `nop` load all cores can boost to $P1_{HW}$, while for integer loads all cores can run only at P_{base}. Boosting under load can be achieved when other modules are either in P_{slow} or halted. `Mwait` provides the power headroom to automatically boost to P_{turbo}. The manual boosting control allows to run one module in P_{turbo} if the others run at P_{slow}.

2.2 Intel's Turbo Boost

Intel's DVFS implementation is largely similar to AMD's but more hardware-centric and mainly differs in the level of manual control. All cores are in the same frequency and voltage domain but can each have an individual C-state. The P-states are based on increasing multipliers for the stepping of 100MHz, non-predefined ACPI P-states in the opposite order. Our processor supports frequencies from 0.8GHz to 3.9GHz corresponding to 32 P-states that are summarized in Table 3. In TURBO terms, P_{base} corresponds to $P34_{HW}$, leaving 5 boosted P-states. All active cores in C0 symmetrically run at the highest requested frequency, even if some cores requested slower P-states. The consumed power was measured in a fashion analogous to that in Section 2.1, with hyper-threading enabled and all cores always in the same P-State.

The processor enables Turbo Boost if not all cores are in C0. The level of boosting depends on the number of active cores, estimated power consumption, and additionally the temperature of the package. This "thermal boosting" allows the processor to temporarily exceed the TDP using the thermal capacitance of the package. In contrast to AMD, the maximum achievable frequency also depends on the recent execution history, which relates to the current package temperature and makes it somewhat stateful. While boosting can be enabled or disabled altogether, the boosted P-states are always controlled automatically by the processor and no manual control by software is possible.

Intel's design choice targets to speed up critical periods of computation, e.g., boosting sequential bottlenecks by putting waiting cores to sleep using C-states or providing temporarily peak performance for interactive applications as on mobile devices or desktops. Our focus is on multi-threaded applications mostly found on servers that run for long periods without much idle time. Thermal boosting is not applicable to such workloads because on average one cannot exceed the TDP. Instead, our goal is to improve the performance within the TDP limits.

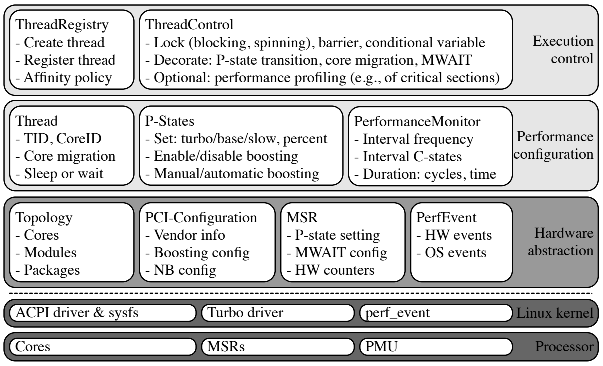


Figure 2: Overview of TURBO *library components.*

3 TURBO Library

The TURBO library, written in C++ for the Linux OS, provides components to configure, control, and profile processors from within applications. Our design goals are twofold: we want to provide a set of abstractions to (1) make it convenient to improve highly optimized software based on DVFS; and (2) set up a testbed for algorithms that explore challenges of future heterogeneous cores [2], such as schedulers. The components of the TURBO library are organized in layers with different levels of abstraction as shown in Figure 2. All components can be used individually to support existing applications that use multiple threads or processes. The layered architecture allows an easy extension to future hardware and OS revisions.

3.1 Processor and Linux Kernel Setup

The default configurations of the processors and Linux kernel manage DVFS transparently for applications: All boosted P-states are controlled by the processor and the Linux governor will adapt the non-boosted P-states based on the current processor utilization ("ondemand") or based on static settings that are enforced periodically ("performance", "userspace").

We must disable the influence of the governors and the processor' power saving features in order to gain explicit control of the P-states and boosting in user space using our library. Note that the "userspace" governor provides an alternative but inefficient P-state interface [16]. Therefore, we disable the CPU frequency driver (`cpufreq`) and turn off AMD's *Cool'n'Quiet* speed throttling technology in the BIOS. To control all available P-states in user space, we can either disable automatic boosting altogether, which is the only solution for Intel, or for AMD set $\#P_{boosted} = 0$ to enable manual boosting control (for details see Section 2). Changing the number of boosted P-states also changes the frequency of the *time stamp counter* (`tsc`) for AMD processors so we therefore disable `tsc` as a clock source for the Linux kernel and instead use the *high precision event timer* (`hpet`). Note that these tweaks can easily be applied to production systems because we only change BIOS settings and kernel parameters.

The processor additionally applies automatic frequency scaling for the integrated NB that can have a negative impact on memory access times for boosted processor cores.

Therefore, NB P-states are disabled and it always runs at the highest possible frequency.

Linux uses the `monitor` and `mwait` instructions to idle cores and change their C-state. When another core writes to the address range specified by `monitor`, then the core waiting on `mwait` wakes up. The `monitor-mwait` facility provides a "polite" busy-waiting mechanism that minimizes the resources consumed by the waiting thread. For experiments on AMD, we enable these processor instructions for user space and disable the use of `mwait` in the kernel to avoid lockouts. Similarly, we must also disable the use of the `hlt` instruction by the kernel, because otherwise we cannot guarantee that at least one core stays in C0. We restrict the C-state for the Linux kernel to C0 and use the polling idle mode. These changes are required in our prototype only for the evaluation of C-state transitions and are not necessary in a production system.

The presented setup highlights the importance of the configuration of both hardware and OS for sound benchmarking. Multi-threaded algorithms should be evaluated by enforcing P_{base} and C0 on all cores to prevent inaccuracies due to frequency scaling and transition latencies. All other sources of unpredictability should be stopped, e.g., all periodic `cron` jobs.

3.2 Performance Configuration Interface

The library must be aware of all threads even if they are managed explicitly by the application. Therefore, the *thread registry* is used first to create or register all *threads*. Next, the threads are typically assigned to distinct cores based on the processor's *topology*, which is discovered during initialization. If thread migration to another core is required at runtime, it must be performed using our library to allow an update of the core specific configuration, e.g., the P-state.

The easiest way to benefit from DVFS is to replace the application's locks with *thread control* wrappers that are decorated with implicit P-state transitions, e.g., boosting the lock owner at P_{turbo}, waiting at P_{slow}, and executing parallel code at P_{base}.

If the wrappers are not sufficient, the application can request an explicit *performance configuration* that is still independent of the underlying hardware. Threads can request the executing core to run at P_{turbo}, P_{base}, or P_{slow}, and can alternatively specify the *P-state* in percent based on the maximum frequency. The actual P-state is derived from the selected setup, e.g., if boosting is enabled and controlled manually. The current P-state configuration is cached in the library in order to save the overheads from accessing the MSRs in kernel space. If a P-state is requested that is already set or cannot be supported by the processor's policy or TDP limits, then the operation has no effect.[2] Threads can also request to temporarily migrate to a dedicated processor core that runs at the highest possible frequency and stays fully operational in C0.

The lowest layer presents *hardware abstractions* for the machine specific interfaces and DVFS implementations, as well as the Linux OS. The Linux kernel provides a device driver that lets applications access MSRs as files under root privilege using `pread` and `pwrite`. We implemented a lightweight TURBO *kernel driver* for a more streamlined access to the processor's MSRs using `ioctl` calls. The driver essentially provides a wrapper for the `wrmsr/rdmsr` instructions to be executed on the current core. Additionally, it allows kernel space latency measurements, e.g., for P-state transition time, with more accuracy than from user space. We derive the *topology* from the Linux ACPI driver and use `sysfs` for AMD's package configuration using PCI functions.

3.3 Performance and Power Profiling

The TURBO library provides means to profile highly optimized applications and algorithms for heterogeneous cores. The profiling can be used to first identify sections that can benefit from frequency scaling and later to evaluate the performance and power implications of different configurations.

Again, the simplest ways to obtain statistics is to use *thread control* wrappers, which exist to replace locks, barriers, and condition variables. The wrappers can be decorated with profiling capabilities of the *performance monitor*, which uses the `aperf/mperf` and `tsc` counters of the processor [1, 19] and the `perf_event` facilities of the Linux kernel to access the processor's *performance monitoring unit* (PMU).

The performance monitor operates in intervals, e.g., defined by a lock wrapper, for which it captures the cycles, frequency, and C-state transitions. Additional counters such as the number of cache misses or stalled cycles can be activated, e.g., to analyze the properties of a critical section. The PMU also provides counters to read the *running average power limit* (RAPL) on Intel and the processor power in TDP on AMD.

4 Processor Evaluation

On top of the TURBO library presented in Section 3, we implemented a set of benchmark applications that configure and profile the underlying processor. In this section, we present (1) the static transition latencies introduced by the OS and hardware, (2) the overheads of blocking upon contended locks and when it pays off regarding speed and energy compared to spinlocks, and (3) new static and dynamic P-state transition strategies that optimize spinlocks and allow applications to expose heterogeneous frequencies.

4.1 Hardware Transition Latency

The latency for DVFS results from a combination of OS overhead to initiate a transition and hardware latency to adjust the processor's state. Therefore, we present in Tables 4 (AMD) and 5 (Intel) the overhead for system calls, P-state requests and the actual transition latencies in isolation. Throughout our evaluation, we use a Linux kernel 3.11 that is configured according to Section 3.1. We use only the x86 cores (ALU)

[2] In practice, we write our request in MSR_P_{cmd} and can read from MSR_P_{val} what the CPU actually decided. We can either (a) wait until both MSRs match, i.e., another core makes room in the TDP, (b) return the CPU's decision, or (c) just write and provide best-effort guarantees (default). Deterministic hardware without thermal boosting does not overwrite MSR_P_{cmd}.

	P-State	Mean		Deviation	
Operation	**Transition**	Cycles	ns	Cycles	ns
System call overheads for `futex` and TURBO driver					
syscall(futex_wait_private)	—	1321	330	42	10
ioctl(trb)	—	920	230	14	3
P-state MSR read/write cost using `msr` or TURBO driver					
pread(msr, pstate)	—	3044	761	43	10
ioctl(trb, pstate)	—	2299	574	30	7
pwrite(msr, pstate, P_{base})	$P_{base} \rightarrow P_{base}$	2067	741	110	27
ioctl(trb, pstate, P_{base})	$P_{base} \rightarrow P_{base}$	1875	468	42	10
Hardware latencies for P-state set (`wrmsr`) and transition (`wait`) (kernel space)					
wrmsr(pstate, P_{slow})	$P_{base} \rightarrow P_{slow}$	28087	7021	105	26
wrmsr(pstate, P_{slow}) & wait	$P_{base} \rightarrow P_{slow}$	29783	7445	120	30
wrmsr(pstate, P_{turbo})	$P_{slow} \rightarrow P_{turbo}$	1884	471	35	8
wrmsr(pstate, P_{turbo}) & wait	$P_{slow} \rightarrow P_{turbo}$	226988	56747	84	21
wrmsr(pstate, P_{base}) & wait	$P_{slow} \rightarrow P_{base}$	183359	45839	130	32
wrmsr(pstate, P_{turbo}) & wait	$P_{base} \rightarrow P_{turbo}$	94659	23664	87	21
wrmsr(pstate, P_{base})	$P_{turbo} \rightarrow P_{base}$	23203	5800	36	9
wrmsr(pstate, P_{base}) & wait	$P_{turbo} \rightarrow P_{base}$	24187	6046	139	34
wrmsr(pstate, $P1_{HW}$)	$P_{base} \rightarrow P1_{HW}$	974	234	132	33
wrmsr(pstate, $P1_{HW}$) & wait	$P_{base} \rightarrow P1_{HW}$	94642	23660	136	34
wrmsr(pstate, P_{base}) & wait	$P1_{HW} \rightarrow P_{base}$	24574	6143	138	34
Hardware latencies for C-state transitions (in user space)					
monitor & mwait	—	1818	454	18	4
Software and hardware latency for thread migration					
pthread_setaffinity	—	26728	6682	49	12

Table 4: Latency cost (AMD FX-8120, 100,000 runs).

	P-State	Mean		Deviation	
Operation	**Transition**	Cycles	ns	Cycles	ns
System call overheads for `futex` and TURBO driver					
syscall(futex_wait_private)	—	1431	366	32	8
ioctl(trb)	—	1266	324	64	16
P-state MSR read/write cost using `msr` or TURBO driver					
pread(msr, pstate)	—	2638	775	24	7
ioctl(trb, pstate)	—	2314	680	54	16
pwrite(msr, pstate, P_{base})	$P_{base} \rightarrow P_{base}$	4246	1248	122	35
ioctl(trb, pstate, P_{base})	$P_{base} \rightarrow P_{base}$	3729	1096	72	21
Hardware latencies for P-state set (`wrmsr`) and transition (`wait`) (kernel space)					
wrmsr(pstate,P_{base})	$P_{slow} \rightarrow P_{base}$	44451	13073	131	38
wrmsr(pstate,P_{base}) & wait	$P_{slow} \rightarrow P_{base}$	48937	14393	86	25
wrmsr(pstate,P_{slow})	$P_{base} \rightarrow P_{slow}$	2015	592	61	17
wrmsr(pstate,P_{slow}) & wait	$P_{base} \rightarrow P_{slow}$	58782	17288	65	19
wrmsr(pstate,P_{turbo})	$P_{base} \rightarrow P_{turbo}$	2012	591	44	12
wrmsr(pstate,P_{turbo}) & wait	$P_{base} \rightarrow P_{turbo}$	41451	12191	78	22
Hardware latencies for C-state transitions (in kernel space)					
monitor & mwait C1	—	4655	1369	25	7
monitor & mwait C2	—	36500	10735	1223	359
monitor&mwait C6	—	74872	22021	672	197
Software and hardware latency for thread migration					
pthread_setaffinity	—	12145	3572	81	23

Table 5: Latency cost (Intel i7-4770, 100,000 runs).

and no FPU or MMX/SSE/AVX to preserve the required headroom for manual boosting.

System calls for device-specific input/output operations (`ioctl`) have a low overhead and are easily extensible using the request code parameter. The interface of the TURBO driver (`trb`) is based on `ioctl`, while the Linux MSR driver (`msr`) uses a file-based interface that can be accessed most efficiently using `pread/pwrite`. The difference in speed between `msr` and `trb` (both use `rdmsr/wrmsr` to access the MSRs) results mostly from additional security checks and indirections that we streamlined for the TURBO driver. The cost in time for system calls depends on the P-state, i.e., reading the current P-state scales with the selected frequency, here P_{base}.

Observation 1: *P-state control should be made available through platform-independent application program interfaces (APIs) or unprivileged instructions. The latter would eliminate the latency for switching into kernel space to access platform-specific MSRs but require that the OS's DVFS is disabled.*

We measured the cost of the `wrmsr` instruction that initiates a P-State transition of the current core, as well as the latency until the transition is finished, by busy `waiting` until the frequency identifier of the P-state is set in the status MSR. Both measurements are performed in the TURBO driver, removing the inaccuracy due to system call overheads.

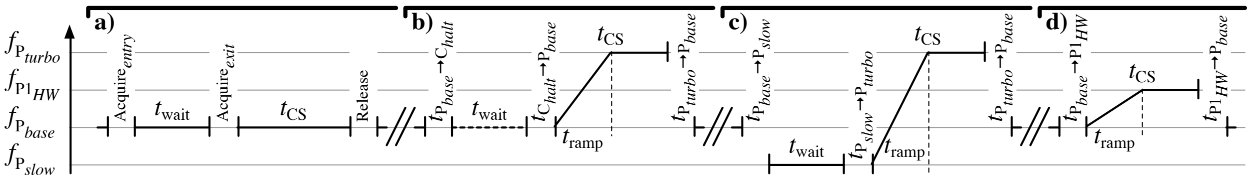
For AMD, requesting a P-state faster than the current one (e.g., $P_{slow} \rightarrow P_{base}$) has low overhead in itself, but the entire transition has a high latency due to the time the VRM takes to reach the target voltage. The request to switch to a slower P-state (e.g., $P_{base} \rightarrow P_{slow}$) has almost the same latency as the entire transition, i.e., the core is blocked during most of the transition. We suspect that this blocking may be caused by a slow handshake to coordinate with the other module's core to see if an actual P-state change will occur. Overall, the transition has a lower latency because the frequency can already be reduced before the voltage regulator is finished. If only switching to a slow P-state for a short period, the transition to a faster P-state will be faster if the voltage was not dropped completely.

On the Intel CPU, total latency results are very similar: A P-state transition also takes tens of microseconds but depends on the distance between the current and requested P-state. A significant difference to AMD, however, lies in the faster execution of the `wrmsr` request of a P-state transition going slower (e.g., $P_{base} \rightarrow P_{slow}$) because Intel does not need to perform additional coordination.

Observation 2: *The frequency transitions should be asynchronous, triggered by a request and not blocking, i.e., keeping the core operational. The API should include the ability to read or query P-state transition costs for building a cost model that allows DVFS-aware code to adapt at runtime.*

We additionally show costs related to the OS. In the `mwait` experiment, one core continuously updates a memory location while the other core specifies the location using `monitor` and calls `mwait`. The core will immediately return to execution because it sees the memory location changed, so the numbers represent the minimal cost of executing both instructions. Although AMD allows the use of `mwait` from user space, the feature is typically used by the OS's `futex` system call when the kernel decides to idle. The `pthread_setaffinity` function migrates a thread to a core with a different L2 Cache that is already in C0 state and returns when the migration is finished. Thread migration typically results in many cache misses but the benchmark keeps only minimal data in the cache.

Observation 3: *The OS should keep the current frequency in the thread context to better support context switches and thread migrations. Ideally, the OS would expose a new set of advisory platform-independent APIs to allow threads to set their desired DVFS-related performance targets. Furthermore, the OS kernel (and potentially a virtual machine hypervisor) would moderate potentially conflicting DVFS resource requests from independent and mutually unaware applications.*



$f_{P_{turbo}}$
$f_{P1_{HW}}$
$f_{P_{base}}$
$f_{P_{slow}}$
a)
Acquire$_{entry}$
t_{wait}
Acquire$_{exit}$
t_{CS}
Release
b)
$t_{P_{base}\rightarrow C_{halt}}$
t_{wait}
$t_{C_{halt}\rightarrow P_{base}}$
t_{ramp}
t_{CS}
$t_{P_{turbo}\rightarrow P_{base}}$
c)
$t_{P_{base}\rightarrow P_{slow}}$
t_{wait}
$t_{P_{slow}\rightarrow P_{turbo}}$
t_{ramp}
t_{CS}
$t_{P_{turbo}\rightarrow P_{base}}$
d)
$t_{P_{base}\rightarrow P1_{HW}}$
t_{ramp}
t_{CS}
$t_{P1_{HW}\rightarrow P_{base}}$

Figure 3: Frequency sequence for (a) spinning, (b) blocking, (c) frequency scaling and (d) critical regions.

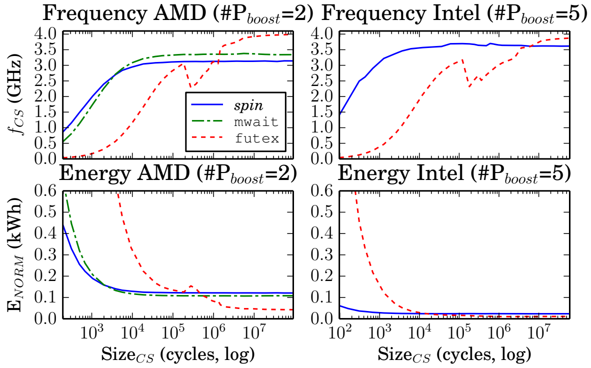


Figure 4: Characteristics of blocking and spinning.

4.2 Blocking vs. Spinning Locks

We evaluate the boosting capabilities using a thread on each core that spends all its time in critical sections (CS). The CS is protected by a single global lock implemented as an MCS queue lock [29] in the TURBO library. The lock is decorated such that upon contention, the waiting thread either *spins* or blocks using `mwait` (AMD only) or `futex`. The sequence is illustrated in Figure 3a and 3b, respectively. In all cases, the thread-local MCS node is used for the notification of a successful lock acquisition. Inside the CS, a thread-local counter is incremented for a configurable number of iterations (~10 cycles each). While the global lock prevents any parallelism, the goal of the concurrent execution is to find the CS length that amortizes the DVFS cost.

We want to discuss when blocking is preferable over spinning, both in terms of performance and energy, using the default configuration of hardware and OS: The P-states are managed automatically by the processor and the setup from Section 3.1 is not applied. We run the application for 100 seconds and count the number of executed CS, which gives us the cycles per CS including all overheads. Separately, we measure the cycles per CS without synchronization at P$_{base}$, i.e., the cycles doing real work. The effective frequency inside a CS is: $f_{CS} = f_{base} * \frac{cycles_{nosync}}{cycles_{mcs}}$. The energy results are based on the processor's TDP/RAPL values, from which we take samples during another execution. We compute the energy it takes to execute 1 hour of work at P$_{base}$ inside CS: $E = E_{sample} * \frac{cycles_{mcs}}{cycles_{nosync}}$.

The results are shown in Figure 4. The *spin* strategy runs all cores at P$_{base}$ and is only effected by synchronization overhead, with decreasing impact for larger sizes of CS. The `mwait` and `futex` strategies are additionally effected by C-state transitions that halt the core while blocking, which allows to boost the active core. The C-state reached by `mwait` is not deep enough to enable P$_{turbo}$, probably because it is requested from user space. Still, CS are executed at P1$_{HW}$ and the low overhead lets `mwait` outperform *spin* already at a CS size of ~4k cycles. Using `futex` has the highest overhead because it is a system call. The C-state reached depends on t_{wait} (see Figure 3b), which explains the performance drop: Deep C-states introduce a high latency (see Table 5) but are required to enable P$_{turbo}$. We verified this behavior using `aperf/mperf`, which showed that the frequency in C0 is at P$_{turbo}$ only after the drop. The `futex` outperforms *spin* and `mwait` at ~1.5M cycles for AMD and ~4M cycles for Intel, which also boosts *spin* 2 steps. Note that an optimal synchronization strategy for other workloads also depends on the conflict probability and t_{wait}, but our focus is on comparing boosting initiated by the processor and on application-level.

The sampled power values do not vary for different sizes of CS (see Tables 2 and 3 for *ALU* and `mwait`), except for `futex`, which varies between 55-124W for AMD depending on the reached C-state. The reduction in energy consumption due to deeper C-states must first amortize the introduced overhead before it is more efficient than spinning. With only a single core active at a time, `futex` is the most energy efficient strategy for AMD after a CS size of ~1M cycles, which results for 8 threads in t_{wait} =~7M cycles because the MCS queue lock is fair. Intel is already more energy efficient after ~10k cycles, indicating that it trades power savings against higher latencies. Boosting provides performance gains for sequential bottlenecks and halting amortizes the active cores' higher energy consumption [31]. The default automatic boosting is not energy efficient for scalable workloads because all energy is consumed only by a single core without performance benefit [12].

4.3 Application-level P-state Transition Strategies

Our goal is to enable application-level DVFS while keeping all cores active. Therefore, we enable manual P-state control with the setup described in Section 3.1 and restrict the following discussion to just AMD. For the evaluation, we use the same application as in the previous Section 4.2 but with a different set of decorations for the lock: The strategy `one` executes iterations only on a single core that sets the P-state statically during initialization to either P$_{slow}$, P$_{base}$ or P$_{turbo}$. All other threads run idle on cores at P$_{slow}$ in C0. This provides the baseline for different P-state configurations without P-state transition overheads but includes the synchronization. The dynamic strategies `ownr` and `wait` are illustrated in Figure 3c. For `ownr`, all threads are initially set to P$_{slow}$ and the lock owner dynamically switches to P$_{turbo}$ during the CS. For `wait`, all

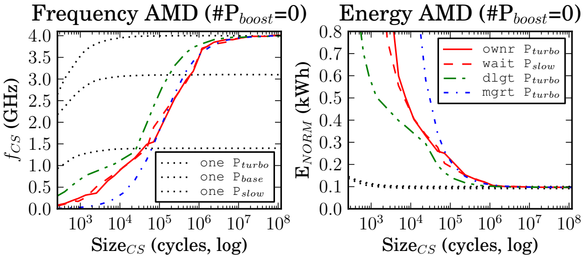


Figure 5: Characteristics of manual P-state control.

threads initially request P$_{turbo}$ and dynamically switch to P$_{slow}$ while waiting. The processor prevents an oversubscription and allows P$_{turbo}$ only if 3 modules are in P$_{slow}$. The remaining strategies use only a subset of the cores for executing CS: `dlgt` uses only 1 thread per module and delegates the P-state transition request to the thread executing on the neighboring core. The strategy is otherwise the same as `ownr`. `mgrt` uses only 6 cores on 3 modules running at P$_{slow}$. The remaining module runs at P$_{turbo}$ and the current lock owner migrates to a core of the boosted module during the CS.

The results are presented in Figure 5. The dynamic strategies `ownr` and `wait` introduce overhead in addition to the synchronization costs because two P-state transitions must be requested for each CS. This overhead is amortized when the resulting effective frequency of the CS is above `one` with P$_{base}$, starting at CS sizes of ∼600k cycles. Both strategies behave similarly because the application does not execute parallel code between CS. Otherwise, the idea is that `wait` hides the slow blocking transition to P$_{slow}$ (see Section 4.1) within t_{wait}, whereas `ownr` must perform this transition after releasing the lock. To that extent, `dlgt` shifts the P-state transition cost entirely to the other core of the module and can outperform `one` already at ∼200k cycles, but only half of the processor cores can be used. The `mgrt` strategy does not include overhead from P-state transitions but costly thread migrations. Still, it outperforms `one` at ∼400k cycles. A real-world benchmark would show worse results because it suffers from more cache misses on the new processor core than our synthetic benchmark that keeps only little data in the cache [30]. Additionally, initiating a migration at P$_{slow}$ will be executed slowly until the thread reaches the boosted core. Overall, we observe that application-level DVFS is more effective than C-state control because it allows to outweigh overheads for CS of sizes smaller than ∼1.5M cycles.

Observation 4: *The P-state transition should be as fast as possible so that short boosted sections can already amortize the transition cost. It exists hardware that can switch to arbitrary frequencies within one clock cycle [17].*

As long as one modules runs at P$_{turbo}$, which is the case here, the processor consumes the maximal TDP of 125W. The consumed energy solely depends on the overheads of each strategy because of the serialized execution. Note that the energy for executing `one` with a static P-state is almost identical for P$_{slow}$, P$_{base}$ and P$_{turbo}$, indicating that the energy consumption is proportional to the P-state. In fact, we get for a single module in P$_{turbo}$ 29% more speed using 25% more power compared to P$_{base}$ (see Table 2). Compared to `mwait` and `futex`, application-level DVFS allows less power savings because all cores stay in C0, but it can be applied to parallel workloads, which we investigate in Section 5.

Observation 5: *Processors should support heterogeneous frequencies individually for each core to provide headroom for boosting while staying active. The design should not limit the frequency domain for a package (Intel) or module (AMD). An integrated VRM supports fine grained voltage domains to allow higher power savings at low speeds. Additionally, for some workloads it would be beneficial to efficiently set remote cores to P_{slow} in order to have local boosting control.*

4.4 Performance Cost Model

Based on our experimental results, we derive a simplified cost model for AMD's boosting implementation to guide developers when boosting pays off regarding performance. We first present a model for boosting sequential bottlenecks that formalizes the results from Section 4.3. We then specialize it for boosting CS that are not a bottleneck as well as for workloads that contain periods with heterogeneous workload distributions.

We make the following simplifying assumptions: (1) the application runs at a constant rate of instructions per cycle (IPC), regardless of the processor frequency; (2) we do not consider costs related to thread synchronization; (3) the frequency ramps linearly towards faster P-states (e.g., $f_{P_{slow}} \rightarrow f_{P_{turbo}}$); and (4) the frequency transition to a slower P-state takes as long as the P-state request. Assumption (4) is a direct result of our latency measurement, (1) and (2) allow an estimation without taking application specifics into account. We will revisit assumptions (1) and (2) when looking at actual applications that depend on memory performance and thus exhibit varying IPC with changing frequency (due to the changed ratio of memory bandwidth, latency and operation frequency).

For sequential bottlenecks, we follow the strategy `ownr` described in Section 4.3 and illustrated in Figure 3c. Boosting will pay off if we outperform the CS that runs at $f_{P_{base}}$: $t_{CS_{f_{P_{turbo}}}} \leq t_{CS_{f_{P_{base}}}}$. The minimal t_{CS} must be greater than the combined P-state request latencies and the number of cycles that are executed during the P-State transition (t_{ramp}, i.e., the difference between `wrmsr` and `wait` in Table 4) to P$_{turbo}$:

$$t_{CS} \geq t_{P_{slow} \rightarrow P_{turbo}} + t_{ramp} + t_{P_{tubo} \rightarrow P_{base}} + \frac{cycles_{CS} - cycles_{ramp}}{f_{P_{turbo}}}$$

Based on the P-state transition behavior that we observed in Section 4.3, we can compute the minimal t_{CS} as follows:

$$t_{CS} \geq \frac{f_{P_{turbo}}}{f_{P_{turbo}} - f_{P_{base}}} \cdot \left(t_{P_{slow} \rightarrow P_{turbo}} + t_{P_{turbo} \rightarrow P_{base}}\right) + \frac{1}{2} \cdot \frac{f_{P_{turbo}} - f_{P_{slow}}}{f_{P_{turbo}} - f_{P_{base}}} \cdot t_{ramp}$$

The minimal wait time t_{wait} to acquire the lock should simply be larger than the time to drop to $f_{P_{slow}}$: $t_{wait} \geq t_{P_{base} \rightarrow P_{slow}}$. With the results from Section 4.1, on AMD this equals to a minimal t_{CS} of ∼436,648 cycles (∼109μs). Note that optimized strategies can reach the break even point already earlier (e.g., `dlgt` in Figure 5). Based on the above cost model for sequential bottlenecks, we can derive a cost model for boosting CS by one step (see Figure 3d):

$$t_{CS} \geq \frac{f_{P1_{HW}}}{f_{P1_{HW}} - f_{P_{base}}} \cdot (t_{P_{base} \rightarrow P1_{HW}} + t_{P1_{HW} \rightarrow P_{base}}) + \frac{1}{2} \cdot t_{ramp}$$

We never move below P$_{base}$ and boosting pays off if t_{CS} is longer than ∼336,072 cycles (∼84μs).

Besides boosting sequential bottlenecks, another interesting target are periods of heterogeneous workload distributions. These workloads can run one thread temporarily at a higher priority than other active threads or have an asymmetric distribution of accesses to CS from threads. Typically, such critical regions are longer because they combine several CS, thus improving the chances of amortizing the transition cost. Based on the presented cost model, we compute the minimal duration of such periods instead of the CS size. We present examples in Section 5.

5 Boosting Applications

We evaluated the TURBO library using several real-world applications with user space DVFS on the AMD FX-8120. We chose these workloads to validate the results from our synthetic benchmarks and the cost model to boost sequential bottlenecks (5.1); highlight gains by using application knowledge to assign heterogeneous frequencies (5.2); show the trade-offs when the IPC depends on the core frequency, e.g., due to memory accesses (5.3); and outweigh the latency cost of switching P-states by delegating critical sections to boosted cores (5.4).

5.1 Python Global Interpreter Lock

The Python Global Interpreter Lock (GIL) is a well known sequential bottleneck based on a blocking lock. The GIL must always be owned when executing inside the interpreter. Its latest implementation holds the lock by default for a maximum of 5ms and then switches to another thread if requested. We are interested in applying some of the P-state configuration strategies presented in Section 4.3 to see if they provide practical benefits. For this evaluation, we use the `ccbench` application that is included in the Python distribution (version 3.4a).

The benchmark includes workloads that differ in the amount of time they spent holding the GIL: (1) the *Pi calculation* is implemented entirely in Python and spends all its time in the interpreter; (2) the computation of *regular expressions* (Regex) is implemented in C with a wrapper function that does not release the GIL; and (3) the *bz2 compression* and *SHA1 hashing* have wrappers for C functions that release the GIL, so most time is spent outside the interpreter. Table 6 summarizes the characteristics of the workloads.

	1 Thread		2 Threads			4 Threads		
Task	python	native	wait	python	native	wait	python	native
`Pi` (P)	72694	160	4919	4933	14	14735	4958	18
`Regex` (C)	116593	160	5533	5556	18	16763	5600	18
`bz2` (C)	17	991	10	24	992	34	25	998
`SHA1` (C)	6	386	8	12	386	11	12	386

Table 6: `ccbench` characteristics: average time (µs) per iteration spent in interpreter (python), executing native code without GIL (native) and waiting for GIL acquisition (wait).

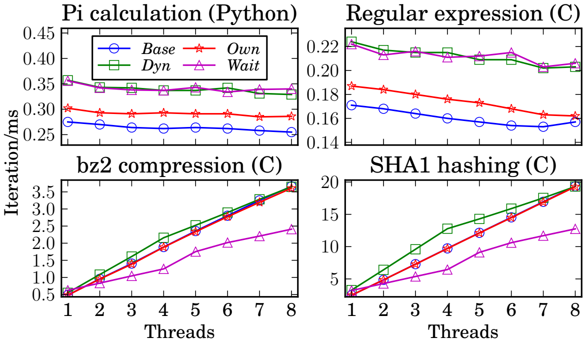


Figure 6: `ccbench` throughput (AMD FX-8120).

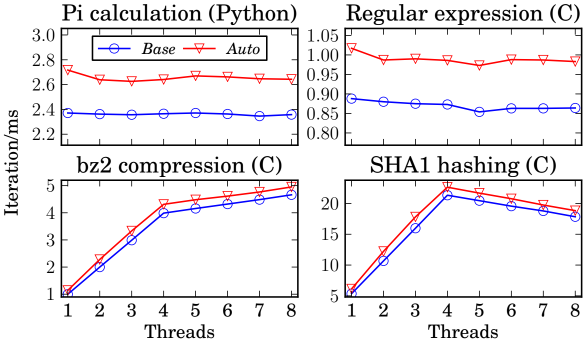


Figure 7: `ccbench` throughput (Intel i7-4770).

We evaluate the following P-state configuration strategies in Figure 6. *Base* runs at P$_{base}$ and, hence, does not incur P-state configuration overheads. *Dyn* waits for the GIL at P$_{slow}$, then runs at P$_{turbo}$ while holding the GIL and switches to P$_{base}$ after releasing it. While the workloads `Pi` and `Regex` do not scale, *Dyn* supports at least the execution at P$_{turbo}$. The performance and power implications are in line with our synthetic benchmark results (Section 4.3) and the cost model (python in Table 6 greater than t_{CS} in Section 4.4). For the workloads `bz2` and `SHA1`, the performance benefit reaches its maximum at 4 threads because we pin the threads such that each runs on a different module, giving the thread full P-state control. When two threads run on a module, more P-state transitions are required per package that eliminate the performance benefit at 8 threads. *Own* runs all threads at P$_{base}$ and boosts temporarily to P1$_{HW}$ while holding the GIL. This manifests in a higher throughput when the GIL is held for long periods but for `bz2` and `SHA` the cost of requesting a P-state transition is not amortized by the higher frequency. *Wait* runs at P$_{turbo}$ if permitted by the TDP and only switches to P$_{slow}$ while waiting for the GIL. This strategy works well with high contention but introduces significant cost if the waiting period is too short (see Table 6).

In Figure 7 we compare Intel's results for boosting disabled (*Base*) and enabled automatically by the processor (*Auto*). Overall, the results are similar to the ones obtained on AMD and what we expect from Section 4.2: The level of boosting depends on the number of halted cores, which enables P$_{turbo}$

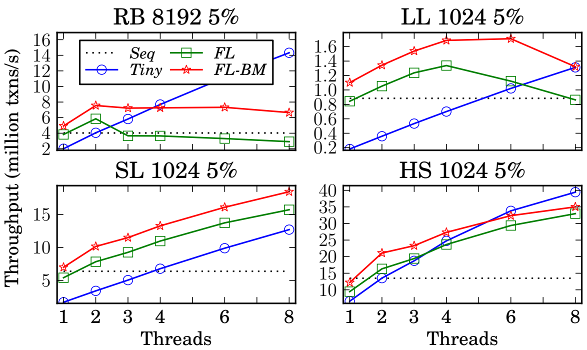


Figure 8: FastLane STM integer set benchmarks.

	RB			**LL**			**SL**			**HS**		
Nb. threads	*2*	*4*	*6*	*2*	*4*	*6*	*2*	*4*	*6*	*2*	*4*	*6*
FL	63	44	35	68	48	44	68	39	24	56	25	13
FL-BM	64	55	54	70	49	53	68	42	28	56	29	16

Table 7: Commit ratio of the master thread (% of all commits).

for `Pi` and `Regex`. `SHA1` and `bz2` boost slightly because not all processor features are used. The performance drop beyond 4 threads is due to hyper-threading.

5.2 Software Transactional Memory

FastLane [43] is a software transactional memory (STM) implementation that processes a workload asymmetrically. The key idea is to combine a single fast master thread that can never abort with speculative helper threads that can only commit if they are not in conflict. The master thread has a very lightweight instrumentation and runs close to the speed of an uninstrumented sequential execution. To allow helper threads to detect conflicts, the master thread must make the in-place updates of its transactions visible (by writing information in the transaction metadata). The helpers perform updates in a write-log and commit their changes after a validation at the end of the transaction. The benefit is a better performance for low thread counts compared to other state-of-the art STM implementations (e.g., TinySTM [13]) that suffer from instrumentation and bookkeeping overheads for scalability.

We used integer sets that are implemented as a *red-black tree* (RB), a *linked list* (LL), a *skip list* (SL), or a *hash set* (HS) and perform random queries and updates [13]. The parameters are the working set size and the update ratio. Either all threads run at P_{base} (*FL*) or the master statically runs at P_{turbo} (*FL-BM*) and the helpers at P_{slow}, except the helper running on the same module as the master. Note that the master thread is determined dynamically. Moreover, we compare with TinySTM (*Tiny*) and uninstrumented sequential execution (*Seq*) at P_{base}. Our evaluation on the AMD processor shows in Figure 8 that running the master and helpers at different speeds (*FL-BM*) enables high performance gains compared to running all threads at P_{base} (*FL*). The higher throughput can outweigh the higher power (50% vs. 2% for LL), thus, being more energy efficient. *Tiny* wins per design for larger thread counts. Table 7 shows that the master can asymmetrically process more transactions at P_{turbo}. While the helpers at P_{slow} can have more conflicts caused by the master, the conflict rate caused by other slow helpers does not change. Dynamically boosting the commits of the helpers did not show good results because the duration is too short.

Bulk Move	**Strategy**	**Ops/s**	**Resize 10MB** ms	stalled	freq	**Resize 1280MB** ms	stalled	freq
10k	baseline	535k	16	63%	3099	2937	67%	3099
10k	stat resizer	547k	15	82%	3999	2666	88%	4000
10k	dyn resizer	547k	15	81%	3980	2691	87%	3987
10k	dyn worker	535k	18	82%	3971	3155	88%	3982
100	baseline	529k	24	66%	3099	4021	68%	3100
100	stat resizer	540k	22	86%	3999	3647	90%	3999
100	dyn resizer	508k	30	56%	3259	4799	59%	3252
100	dyn worker	461k	48	60%	3211	7970	60%	3265
1	baseline	237k	770	72%	3099	103389	72%	3099
1	stat resizer	245k	721	94%	3999	98056	95%	4000
1	dyn resizer	209k	893	62%	3112	120430	63%	3113
1	dyn worker	90k	1886	64%	3111	252035	65%	3113

Table 8: Memcached hash table resize statistics.

This workload highlights the importance of making P-state configuration accessible from the user space. It allows to expose properties of the application that would otherwise not be available to the processor. For applications that contain larger amounts of non-transactional code, supporting the ability to remotely set P-states for other cores would be very helpful. When a master transaction is executed, it could slow down the other threads in order to get fully boosted for a short period.

5.3 Hash Table Resize in Memcached

Memcached is a high performance caching system based on a giant hash table. While for the normal operation a fine-grained locking scheme is used, the implementation switches to a single global spinlock that protects all accesses to the hash table during the period of a resizing. The resize is done by a separate maintenance thread that moves items from the old to the new hash table and processes a configurable number of buckets per iteration. Each iteration acquires the global lock and moves the items in isolation.

Our evaluation was conducted with Memcached version 1.4.15 and the `mc-crusher` workload generator. We used the default configuration with 4 worker threads that we pinned on 2 modules. The maintenance thread and `mc-crusher` run on their own modules. The workload generator sends a specified number of *set* operations with distinct keys to Memcached, which result in a lookup and insert on the hash table that will eventually trigger several resizes. The hash table is resized when it reaches a size of $2^x \times 10$MB. The cache is initially empty and we insert objects until the 7th resize of $2^7 \times 10$MB (1280MB) is finished.

For the intervals in which the maintenance thread was active, we gathered for the first (10MB) and the last (1280MB) resize interval. These are reported in Table 8: number of items that are moved during one iteration (bulk move, configurable), rate of *set* operations during the entire experiment (ops/s), length of the resize interval (ms), the number of (stalled) instructions and average frequency achieved by the maintenance thread (freq).

We applied the following strategies during the resizing period: *baseline* runs all threads at P_{base}, *stat resizer* runs the maintenance thread at P_{turbo} for the entire period, *dyn resizer* switches to P_{turbo} only for the length of an bulk move iteration and causes additional transition overheads, *dyn worker* switches to P_{slow} while waiting for the maintenance thread's iteration to finish. The last strategy does not show a performance improvement because the cost cannot be amortized especially when the bulk move size gets smaller. The *stat resizer* shows the best performance because it reduces the resizing duration.

While the benchmark shows the benefit of assigning heterogeneous frequencies, an interesting observation is that the speedup achieved by boosting is limited because the workload is mainly memory-bound. Compared to *baseline*, *stat resizer* shows only a speedup of the resize interval between 7%–9% while it runs at a 22% higher frequency. The higher the frequency, the more instructions get stalled due to cache misses that result from the large working set. The number of stalled instructions effectively limit the number of instructions that can be executed faster at a higher frequency. On the other hand, the high cost of the P-state transitions in the dynamic strategy *dyn resizer* is hidden by an decreased number of stalled instructions but it still cannot outweigh the transition latency. Memcached's default configuration performs only a single move per iteration, which according to our results shows the worst overall duration of the experiment (ops/s). A better balance between worker latency and throughput is to set bulk move to 100. With this configuration, memcached spends 15% of its execution time for resizes, which we can boost by 10%. This reduces the total execution time by 1.5% and allows 1.5% more ops/s because the worker threads spent less time spinning. Combined, this amortizes the additional boosting energy.

5.4 Delegation of Critical Sections

We have shown that critical sections (CS) need to be relatively large to outweigh the latencies of changing P-states. Remote core locking [27] (RCL) is used to dedicate a single processor core to execute all application's CS locally. Instead of moving the lock token across the cores, the actual execution of the critical section is delegated to a designated server. We leverage this locality property by statically boosting the RCL server and eliminate the P-state transition overhead for small CS.

We experiment with three of the SPLASH-2 benchmarks [44] and the accompanying version of BerkeleyDB [33].

We report speedup for all workloads over the single-threaded baseline P-state in Figure 9, and find that we obtain only incremental performance gains for the boosted cases. We show various combinations of worker P-states (reported as "W P*x*") and P-states for the RCL server core ("R P*x*"), and contrast these with configurations where all cores run at P_{base} ("All P2") and $P4_{HW}$ ("All P4") for comparison. Note that we do show standard deviation of 30 trials, but there is hardly any noise visible. We do not reduce the P-state for the waiting workers (due to latency reasons), but it seems there is enough TDP headroom for the brief RCL invocations to run even at $P1_{HW}$ and we get speedups of 4% - 9%. As expected, the relative boost is larger if we start from a lower baseline at $P4_{HW}$. Overall, scalability of the benchmarks is good, reserving one core exclusively for RCL will cap scalability at 7 (worker) threads. The authors of RCL claim, however, that reserving this single core pays off in comparison to cache coherence traffic arising from spinlock ownership migrating between cores.

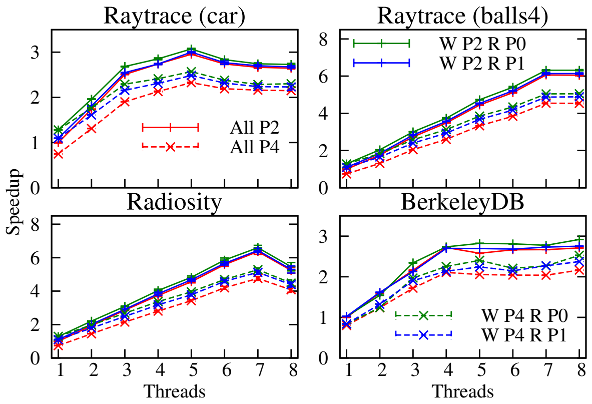


Figure 9: Throughput of SPLASH-2 and BerkeleyDB.

P-state Config	Intra Module 0:0	 100:10	 500:50	Cross Module 0:0	 100:10	 500:50
All P2	91	169	461	480	570	876
W P2 R P1	83	154	421	468	557	906
W P2 R P0	70	131	357	445	491	772
All P4	123	227	621	578	699	1161
W P4 R P1	83	155	421	519	636	1112
W P4 R P0	70	133	358	417	566	1010

Table 9: Core to core memory transfer latency (ns) for an average round-trip (work iterations: N_{Worker} :N_{RCL}, 0.65ns each).

Focusing our attention on the CS, we find them to be short (with a peak at ∼488ns) for the selected benchmarks. To better understand the cost of communication and its behavior under various boosting scenarios, we implemented the core of the RCL mechanism, simple cross-thread polling message passing with two threads, in a small micro-benchmark. We report results for select configurations in Table 9 for AMD, which reflect unloaded latency with no competition for communication channels. Overall we were surprised by the round-trip delay when crossing modules, 480ns, vs. 91ns when communicating inside a module (both at P_{base}). Intra-module communication benefits greatly from boosting (91ns vs. 70ns), due to both communication partners and the communication link (shared L2 cache) being boosted. Communicating cross-module, boosting has a smaller performance impact on the communication latency (480ns vs. 445ns, via L3 cache), which helps to explain the small benefit seen in our workloads with short CS.

6 Related Work

The field of DVFS is dominated by work about improving energy efficiency [23, 32, 14]. DVFS is proposed as a mid-term solution to the prediction that, in future processor generations, the scale of cores will be limited by power constraints [11, 7, 2]. In the longer term, chip designs are expected to combine few large cores for compute intensive tasks with many small cores for parallel code on a single heterogeneous chip. Not all cores can be active simultaneously due to thermal constraints [42, 22]. A similar effect is achieved by introducing heterogeneous voltages and frequencies to cores of the same ISA [10]. Energy efficiency is achieved by reducing the frequency and it was observed that the overall performance is only reduced slightly because it is dominated by memory [25] or network latencies.

Semeraro *et al.* [40] propose multiple clock domains with individual DVFS. Inter-domain synchronization is implemented using existing queues to minimize latency, and frequency can be reduced for events that are not on the application's critical path. The energy savings can be extended by profile-based reconfiguration [28, 5]. Another interesting approach to save power is to combine DVFS with inter-core prefetching of data into caches [21]. This can improve performance and energy efficiency, even on serial code, when more cores are active at a lower frequency. Choi *et al.* [8] introduce a technique to decompose programs into CPU-bound (on-chip) and memory-bound (off-chip) operations. The decomposition allows fine tuning of the energy-performance trade-off, with the frequency being scaled based on the ratio of the on-chip to off-chip latencies. The energy savings come with little performance degradation on several workloads running on a single core. Hsu *et al.* [18] propose an algorithm to save energy by reducing the frequency with HPC workloads. Authors also present and discuss transition latencies. A recent study [24] on the Cray XT architecture, which is based on AMD CPUs, demonstrates that significant power savings can be achieved with little impact on runtime performance when limiting both processor frequency and network bandwidth. The P-states are changed before the application runs. It is recommended that future platforms provide DVFS of the different system components to exploit the trade-offs between energy and performance. Our work goes in the same direction, by investigating the technical means to finely control the states of individual cores.

While energy efficiency has been widely studied, few researchers have addressed DVFS to speed up workloads [15]. Park *et al.* [34] present a detailed DVFS transition overhead model based on a simulator of real CPUs. For a large class of multi-threaded applications, an optimal scheduling of threads to cores can significantly improve performance [37]. Isci *et al.* [20] propose using a lightweight global power manager for CPUs that adapts DVFS to the workload characteristics. Suleman *et al.* [41] optimize the design of asymmetric multi-cores for critical sections. A study of Turbo Boost has shown that achievable speedups can be improved by pairing CPU intensive workloads to the same core [6]. This allows masking delays caused by memory accesses. Results show a correlation between the boosting speedup and the LLC miss rate (high for memory-intensive applications). DVFS on recent AMD processors with a memory-bound workload limits energy efficiency because of an increase of static power in lower frequencies/voltages [26]. Ren *et al.* [38] investigate workloads that can take advantage of heterogeneous processors (fast and slow) and show that throughput can be increased by up to 50% as compared with using homogeneous cores. Such workloads represent interesting use cases for DVFS.

Our TURBO library complements much of the related work discussed in this section, in that it can be used to implement the different designs and algorithms proposed in these papers.

7 Conclusion

We presented a thorough analysis of low-level costs and characteristics of DVFS on recent AMD and Intel multi-core processors and proposed a library, TURBO[3], that provides convenient programmatic access to the core's performance states. The current implementation by hardware and OS is optimized for transparent power savings and for boosting sequential bottlenecks. Our library allows developers to boost performance using properties available at application-level and gives broader control over DVFS. We studied several real-world applications for gains and limitations of automatic and manual DVFS. Manual control exposes asymmetric application characteristics that would be otherwise unavailable for a transparent optimization by the OS. Limitations arise from the communication to memory and other cores that restict the IPC. Our techniques, while useful today, also bring insights for the design of future OS and hypervisor interfaces as well as hardware DVFS facilities.

For the future, we plan to add an automatic dynamic tuning mechanism: based on decorated thread control structures, e.g., locks, we can obtain profiling information and predict the optimal frequency for each core. We also envision use cases beyond optimizing synchronization, such as DVFS for flow-based programming with operator placement (deriving the frequency from the load factor) or data routing (basing DVFS on deadlines or priorities). Finally, the TURBO library provides a research testbed to simulate future heterogeneous multi-core processors with fast/slow cores, as well as to evaluate algorithms targeting energy efficiency or undervolting.

Acknowledgements: We thank André Przywara for his help on AMD's P-states and our shepherd Emmett Witchel. This research has been funded in part by the European Community's Seventh Framework Programme under the ParaDIME Project, grant agreement no. 318693.

References

[1] AMD. BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, 2012.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *ACM CACM*, 2011.

[3] A. Branover, D. Foley, and M. Steinman. AMD Fusion APU: Llano. *IEEE Micro*, 2012.

[4] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen. A dynamic voltage scaled microprocessor system. *IEEE JSSC*, 2000.

[5] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.

[6] J. Charles, P. Jassi, N. Ananth, A. Sadat, and A. Fedorova. Evaluation of the intel core i7 turbo boost feature. In *IISWC*, 2009.

[3] `https://bitbucket.org/donjonsn/turbo`

[7] A. A. Chien, A. Snavely, and M. Gahagan. 10x10: A general-purpose architectural approach to heterogeneity and energy efficiency. *ELSEVIER PCS*, 2011.

[8] K. Choi, R. Soma, and M. Pedram. Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In *DATE*, 2004.

[9] D. Dice, N. Shavit, and V. J. Marathe. US Patent Application 20130047011 - Turbo Enablement, 2012.

[10] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *MICRO*, 2003.

[11] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[12] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking back on the language and hardware revolutions: measured power, performance, and scaling. In *ASPLOS*, 2011.

[13] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.

[14] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *ISPLED*, 2007.

[15] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 2008.

[16] D. Hillenbrand, Y. Furuyama, A. Hayashi, H. Mikami, K. Kimura, and H. Kasahara. Reconciling application power control and operating systems for optimal power and performance. In *ReCoSoC*, 2013.

[17] S. Hoppner, H. Eisenreich, S. Henker, D. Walter, G. Ellguth, and R. Schuffny. A Compact Clock Generator for Heterogeneous GALS MPSoCs in 65-nm CMOS Technology. *IEEE TVLSI*, 2012.

[18] C.-h. Hsu and W.-c. Feng. A power-aware run-time system for high-performance computing. In *SC*, 2005.

[19] Intel. Intel 64 and IA-32 Architectures Software Developers Manual, 2013.

[20] C. Isci, A. Buyuktosunoglu, C.-Y. Chen, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *MICRO*, 2006.

[21] M. Kamruzzaman, S. Swanson, and D. Tullsen. Underclocked software prefetching: More cores, less energy. *IEEE Micro*, 2012.

[22] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *MICRO*, 2003.

[23] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 2005.

[24] J. H. Laros, III, K. T. Pedretti, S. M. Kelly, W. Shu, and C. T. Vaughan. Energy based performance tuning for large scale high performance computing systems. In *HPC*, 2012.

[25] M. Laurenzano, M. Meswani, L. Carrington, A. Snavely, M. Tikir, and S. Poole. Reducing energy usage with memory and computation-aware dynamic frequency scaling. In *Euro-Par*, 2011.

[26] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *HotPower*, 2010.

[27] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In *USENIX ATC*, 2012.

[28] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA*, 2003.

[29] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM TOCS*, 1991.

[30] A. Mendelson and F. Gabbay. The effect of seance communication on multiprocessing systems. *ACM TOCS*, 2001.

[31] 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.

[32] K. Nowka, G. Carpenter, E. MacDonald, H. Ngo, B. Brock, K. Ishii, T. Nguyen, and J. Burns. A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling. *IEEE JSSC*, 2002.

[33] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley db. In *USENIX ATC*, 1999.

[34] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE TCAD*, 2012.

[35] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *MobiCom*, 2001.

[36] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. Computational Sprinting on a Hardware/Software Testbed. In *ASPLOS*, 2013.

[37] B. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu. Cherry-picking: exploiting process variations in dark-silicon homogeneous chip multi-processors. In *DATE*, 2013.

[38] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity for interactive services. In *ICAC*, 2013.

[39] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 2012.

[40] G. Semeraro, G. Magklis, R. Balasubramonian, D. Albonesi, S. Dwarkadas, and M. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *HPCA*, 2002.

[41] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ASPLOS*, 2009.

[42] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS*, 2010.

[43] J.-T. Wamhoff, C. Fetzer, P. Felber, E. Rivière, and G. Muller. Fastlane: improving performance of software transactional memory for low thread counts. In *PPoPP*, 2013.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.

Implementing a Leading Loads Performance Predictor on Commodity Processors

Bo Su[†*], Joseph L. Greathouse[‡], Junli Gu[‡], Michael Boyer[‡], Li Shen[†], Zhiying Wang[†]

[†]State Key Lab of HPC, College of Computer, National University of Defense Technology
[‡]AMD Research

Abstract

Modern CPUs employ Dynamic Voltage and Frequency Scaling (DVFS) to boost performance, lower power, and improve energy efficiency. Good DVFS decisions require accurate performance predictions across frequencies. A new hardware structure for measuring *leading load* cycles was recently proposed and demonstrated promising performance prediction abilities in simulation.

This paper proposes a method of leveraging existing hardware performance monitors to emulate a leading loads predictor. Our proposal, LL-MAB, uses existing miss status handling register occupancy information to estimate leading load cycles. We implement and validate LL-MAB on a collection of commercial AMD CPUs. Experiments demonstrate that it can accurately predict performance with an average error of 2.7% using an AMD Opteron^{TM}4386 processor over a 2.2x change in frequency. LL-MAB requires no hardware- or application-specific training, and it is more accurate and requires fewer counters than similar approaches.

1 Introduction

Dynamic voltage and frequency scaling (DVFS) is used to optimize performance under power and energy constraints, typically under the control of the OS or firmware. One of the key challenges of utilizing DVFS effectively is dynamically predicting the performance impact of frequency changes for arbitrary applications. This can be difficult because program execution time does not depend solely on core frequency. While some sections of a program will run faster at higher frequencies, others are limited by non-core components, such as DRAM latency. Because of this, simple linear scaling models (where performance is directly proportional to frequency) often yield inadequate estimates [5].

Unfortunately, it can be difficult to predict the effect of memory accesses on performance. Not all memory accesses cause stalls to the core because of caches in the core clock domain. In addition, processors can exploit memory-level parallelism (MLP) by overlapping multiple cache misses. As such, not all cache misses directly affect the performance. Finally, DRAM access latency varies with access patterns, making it difficult to predict time spent waiting on memory from access counts alone.

One promising approach for predicting DVFS performance is the recently proposed *leading loads* model [5, 7, 10], which splits the execution time of an application into time spent in the core (which changes along with frequency) and in the memory (which does not). It then uses new leading load performance counters to accurately estimate this memory time. Simulations demonstrated that this approach could predict application performance under DVFS with an order of magnitude higher accuracy than previously proposed models, while requiring no training (unlike regression-based models). These results led the authors to suggest that hardware support for leading loads should be added to future processors.

This paper demonstrates how to leverage *existing* hardware performance counters that measure miss status handling register (MSHR) activity on commodity AMD processors in order to approximate a leading loads performance predictor. This predictor can accurately estimate the performance impact of DVFS changes on arbitrary applications running on real hardware. It requires no hardware- or application-specific training, and uses only a small number of performance counters.

We validate our method on three different AMD processors across multiple hardware generations. We compare our technique with previously proposed predictors and explain how it is different from an ideal leading load predictor. We show that our model provides a more accurate prediction with less variance in error rate than other predictors that work on existing hardware. To the best of our knowledge, this is the first time the leading loads model has been demonstrated on real hardware.

[*] This work took place while Bo Su interned with AMD Research.

2 Related Work

There is a large body of work on performance modeling under frequency variation. Rountree et al. [10] and Eeckhout [4] describe many of the previous techniques. Eeckhout categorizes analytic performance estimation models into *empirical* models, which use black-box approaches, and *mechanistic* models which are designed from the underlying machine principals. Some of the most popular empirical models are based on regression [11]. They can have good accuracy, but need many input variables and long training runs. Their accuracy is a function of the quality of the training set, and they must be retrained whenever the underlying machine changes.

Mechanistic models often include simplifications and abstractions to make the problem tractable. Proportional scaling models (or "linear scaling") are the simplest and assume that performance scales linearly with frequency. These are simple to implement, but only work well when the application spends little time accessing memory.

Recent mechanistic DVFS estimation models are built from the underlying concept that program execution time is split into *core time* (time doing work) and *memory time* (time stalled waiting for memory). Core time is inversely proportional to frequency. However, because the latency to memory does not change when the core's frequency changes, memory time is not affected by DVFS.

The difficulty of this performance estimation mechanism lies in appropriately characterizing these times. Modern processors can execute instructions out of order, with multiple loads accessing memory in parallel. Simple memory models do not capture this, which leads to incorrect estimates. For example, stall models monitor the amount of time that a core is not committing instructions and assume that this is due to time spent in the memory system [5]. We will show later that this is often an inaccurate assumption, since processors can stall for numerous reasons besides memory latency.

2.1 Leading Loads (LL) Model

Leading loads were simultaneously defined by three groups attempting to solve the problems of these linear and stall models [5, 7, 10]. They utilized the insight that, while many memory accesses may be outstanding, only one can stall the pipeline. As such, the first non-speculative load that misses in the last level of the core's cache is considered a *leading load*. The time between the miss and when it returns is assumed to be memory time. This time is counted even if core work continues under the miss. All misses until this load returns are not leading loads – they represent MLP . A simplifying assumption of this model is that these MLP accesses will not eventually stall the pipeline.

Once a leading load returns, the next miss becomes the leading load. All time when there is no leading load is counted as core time. This is illustrated in Figure 1, where there are 3 load misses: A, B and C. The misses begin at t1, t2, and t6, then finish at t4, t5, and t8, respectively. A and C are leading loads; their delay is memory time. B is not a leading load because a leading load (A) already exists; it is MLP and does not stall the core.

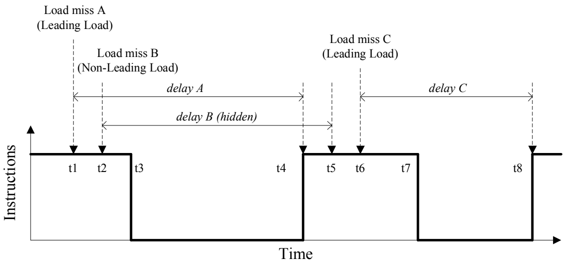


Figure 1: Leading loads example.

There are limitations to this model, such as the assumption that MLP loads will not stall the pipeline, and its inability to deal with bandwidth-bound applications [9]. However, the simulated results for these counters appear promising (with estimation errors of 0.2%), and the hardware is simple, requiring only a single counter per core. The major impediment was the apparent lack of leading load hardware performance events, which prevented testing this mechanism outside of simulation.

2.2 Green Governor (GG) Model

Because their newly proposed leading load counter did not yet exist in hardware, Spiliopoulos et al. also devised a simpler model in their Green Governors work that used existing counters [12]. They monitor the number of last level cache (LLC) misses and number of cycles without a retired instruction. They then characterize the average miss latency using a tool such as lmbench [8] and multiply this delay by the number of LLC misses to estimate memory time. If the amount of time not retiring instructions is less than this, the smaller time is used instead.

When predicting the program's performance from frequency f to f', this model can be described by Equation 1. The memory time M_t is calculated using stall cycles S, the number of LLC misses N, the per-miss delay time D, and the original frequency f. The new execution time T' is then calculated from M_t, the original execution time T, and the ratio of frequency change.

$$T' = (T - M_t) \times \frac{f}{f'} + M_t; M_t = min(\frac{S}{f}, N \times D) \qquad (1)$$

This model makes a number of simplifying assumptions, since it is constrained by existing hardware. First, it assumes that LLC misses have a constant latency, since it only measures miss count. Second, it ignores MLP. Nonetheless, with careful tuning, it can yield reasonably accurate performance estimates.

3 Implementing Leading Loads

On AMD CPUs, a Miss Address Buffer (MAB)[1] is a structure that tracks a single outstanding cache miss. Misses are assigned to available MABs based on a fixed priority; a miss that occurs when the highest priority MAB is available will always be assigned to that MAB. The first miss will be assigned to the highest priority MAB, as will the next miss after that one returns. Thus, the amount of time that the highest priority MAB is occupied represents the aggregate latency of all of the leading loads. This MAB's occupancy time can be measured directly (in clock cycles) using a hardware performance counter. To the best of our knowledge, no such mechanism exists in processors from other vendors.

We thus describe our LL-MAB model, which assumes that the wall-clock occupancy time of the highest priority MAB will remain unchanged across different core frequencies, since its occupancy relies on values returning from memory. The remaining execution time, when nothing is in the highest priority MAB, is core time, which is inversely proportional to frequency. More formally, if an application's execution time is T at frequency f and the highest priority MAB is occupied for M clock cycles, then the predicted execution time T' of the same application at a frequency f' is given by Equation 2.

$$T' = \frac{M}{f} + (T - \frac{M}{f}) \times \frac{f}{f'} \qquad (2)$$

While Family 15h cores measure MAB occupancy time for L2 cache misses, Family 10h cores measure *L1* cache misses. For the purposes of implementing the LL model, this introduces two inaccuracies. First, a leading load from the L1 may still hit in the L2, which is also in the core clock domain. Second, for leading loads that miss in both the L1 and L2 caches, the MAB occupancy time includes the latency of the request from the L1 to L2. Neither should be counted as leading load time, since they will change as the core frequency changes.

In addition, the MABs hold prefetch misses, which should not be counted as leading loads because they will not cause the core to stall. We will show in Section 4 that these inaccuracies are small enough that LL-MAB model is still more accurate than existing predictors.

We implemented LL-MAB on three different AMD processors with two different microarchitectures, described in Table 1. These cores assign MABs in slightly different orders: Family 10h and Family 15h processors give highest priority to MAB1 and MAB0, respectively. Our LL-MAB implementation uses these counters to estimate the leading load time.

We also implemented an enhanced version of the Green Governor (GG) performance estimation model for comparison. Because the L3 cache in AMD processors is in a separate clock domain from the cores, its access time will remain constant at different core frequencies. Spiliopoulos et al. measured *last level cache* misses, which means that they did not measure L2 (core domain) misses that hit in the L3 (memory domain). To compare both of these designs, we build two GG models: one that counts L2 misses, and one that counts L3 misses.

Table 1: Processor configurations and hardware events.

	AMD Phenom^{TM}II X6 1090T	**AMD Opteron^{TM} 4386**	**AMD A10-5800K**
Family	10h	15h	15h
Core Freq.	1.6/3.2 GHz	1.4/3.1 GHz	1.4/3.8 GHz
DRAM	DDR3-800	DDR3-1600	DDR3-1066
MAB Counter	MAB1	MAB0	MAB0
L3 Latency	13.0ns	32.2ns	n/a
L2 Latency	24.7ns	12.8ns	32.3ns
H/W Event	**Event Select Code**		
E1 Exe. Cyc.	0x00410076	0x00410076	0x00410076
E2 MAB Cyc.	0x00410169	0x00410069	0x00410069
E3 Stall Cyc.	0x014100c0	0x014100c0	0x014100c0
E4 L3 Misses	0x4004107e0	0x40040f7e1 0x40040ffe1	No L3 Cache
E5 L2 Misses	0x0001077e	0x00410043	0x00410043

Different memory access patterns cause different DRAM delays. As such, rather than using a single memory latency chosen arbitrarily from a microbenchmark, we instead search the space of possible latencies to find the value that yields the lowest estimation error. This means that we are testing the algorithm on its training data, which may yield optimistic results from the GG model. However, this allows us to operate under the assumption that the GG model's latency has been chosen well (which may not always be the case).

Finally, where possible, our GG models did not use cache misses caused by prefetchers. On Family 15h processors, specifying the appropriate selection of L3 performance events allows us to ignore prefetch misses. However, this is not possible on Family 10h processors.

Table 1 shows the configurations of the systems we measured and the specific performance events we used to collect the data required by the predictors in our experiments. All of the predictors use the Program Cycles counter, though this could potentially be replaced by the hard-coded timestamp counter to reduce counter requirements. The linear estimation method uses only this, while the stall model also uses Stall Cycles. LL-MAB model needs only one additional counter: MAB Wait Cycles. The GG models need 2 or 3 more. GG-L3 uses Stall Cycles and L3 Misses (Family 15h uses 2 hardware counters for this in order to remove prefetch misses). In GG-L2, L3 Misses are replaced by L2 Misses.

[1] Commonly known as a Miss Status Handling Register (MSHR).

4 Evaluation Results and Analysis

4.1 Experimental Methodology

We validated our LL-MAB model using 66 single-threaded benchmarks from the NAS Parallel Benchmarks [1][2], PARSEC [2][3], Rodinia [3][4], and SPEC CPU2006 [6][5]. For comparison, we also tested the linear, stall, GG-L2 and GG-L3 models.

The AMD Phenom^{TM}II 1090T processor ran Canonical Ubuntu Desktop 12.04 (kernel 3.2.0-24), the AMD Opteron^{TM}4386 processor ran Fedora^{TM}19 Desktop (kernel 3.10.6-200), and the AMD A10-5800K processor ran CentOS release 6.4 (kernel 2.6.32-358.23.2). We used msr-tools to set and read the performance counters, cpufreq to change DVFS states, and numactl to lock each benchmark to a single core. For each processor, we measured two frequencies, the higher at least twice the lower. We then estimated the change in user-level unhalted clock cycles between the two frequencies.

As mentioned, we chose the GG miss latency that provided the best average prediction accuracy across all of the benchmarks, based on an exhaustive search of all possible latency values from 0 to 200ns with a step size of 0.1ns. These values are also listed in Table 1. Because the L3 misses on the AMD Phenom^{TM}II 1090T processor include prefetch misses, the ideal L3 miss latency was lower than the L2 miss latency. These values are up to 2× different than those measured with microbenchmarks, which implies that our GG results are optimistic.

4.2 Experimental Evaluation

Figures 2, 3, and 4 show the average and standard deviation of the absolute value of the prediction errors. In this case, error is the difference between the estimated and the actual cycles at one frequency when gathering statistics at the other. Unless noted, all results are listed in the order: AMD Phenom^{TM}II 1090T processor, AMD Opteron^{TM}4386 processor, AMD A10-5800K processor. The stall-based model had much higher average error than the others and is not shown. Its average errors were 21.7%, 31.3% and 32.0%, respectively.

As shown in the figures, our LL-MAB predictor had the lowest average error and standard deviation for all three processors. The GG-L2 and GG-L3 models are slightly worse, but have similar accuracy to one another. It is worth reiterating that their latency values were carefully tuned to reduce error rate, so these numbers do not necessarily mean that one is better than the other. Nonetheless, LL-MAB model's average error was 5.27%, 2.71%, and 4.80%, 1.22, 1.30, and 1.71 percentage points lower than the most accurate GG model. Linear estimation had the worst average prediction error among the models shown.

While a lower prediction error is preferred, the second graph in each of these figures demonstrates the standard deviation of these error rates. In this case, the LL-MAB model had a much smaller variance in its errors, implying a more consistent error rate. This is especially important for performance estimation, since an outlier can lead to a severe loss in performance or energy efficiency.

Figures 5 and 6 plot the absolute prediction error versus memory boundedness for each benchmark. As described by Rountree et al. [10], memory boundedness is the ratio of measured execution cycles at the two frequencies. For compute-bound applications, the number of execution cycles should be (approximately) fixed regardless of the frequency, so the memory boundedness should be (approximately) one. Larger values indicate applications that spend more time in the memory system.

By definition, the linear model's error is proportional to the memory boundedness, so its error rate was highest for memory-bound programs. The stall-based model, on the other hand, exhibited large errors for compute-bound programs, because it incorrectly assumed that compute-bound applications with many pipeline stalls (due to, for example, mispredicted branches) were memory bound.

The GG models use cache miss counts to reduce the prediction error when the memory boundedness is low. In this way, the GG models overcame the high error rate of the stall-based model for more compute-bound applications, while keeping the relatively low error rate of the stall-based model for more memory-bound applications.

Our LL-MAB model was accurate across a range of benchmarks and hardware, because it more directly measures the time spent in the memory system. However, as the memory boundedness increases, the limitations discussed in Section 3 cause some errors. For the programs whose memory boundedness is greater than 1.1, the average absolute error of LL-MAB is still the lowest.

4.3 LL-MAB Model Discussion

LL-MAB demonstrates three primary advantages:

1. LL-MAB provides better average prediction accuracy. This was true despite the fact that we gave our comparison point, GG, as many advantages as possible. The accuracy of the GG models would be even worse using directly measured miss latencies.

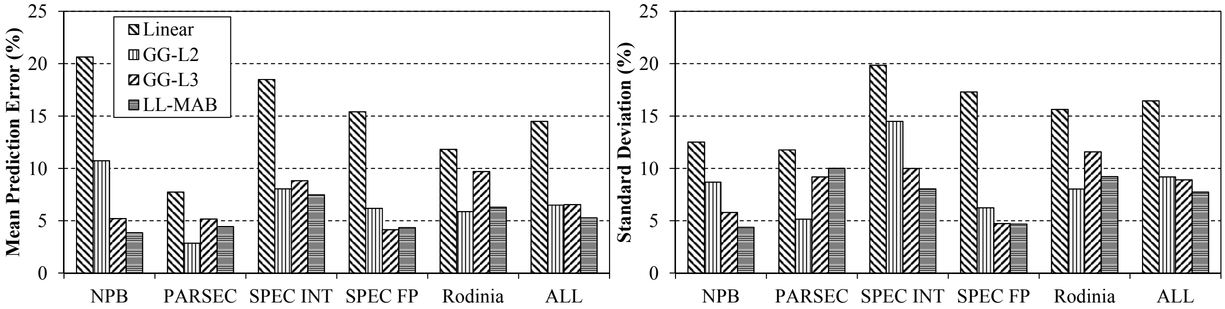
2. LL-MAB is easier to implement. It only requires two performance counters, while the GG models need three or four. The LL-MAB model also requires no hardware- or application-specific training, unlike Green Governor or regression models.

[2] NPB: All 10 *SER* programs; size "B" for DC and "C" for others.

[3] PARSEC: All 13 *gcc-serial* benchmarks with *native* inputs.

[4] Rodinia: bfs, b+tree, heartwall, hotspot, kmeans, lavaMD, leukocyte, lud, particlefilter, pathfinder, srad, cfd, nw, streamcluster.

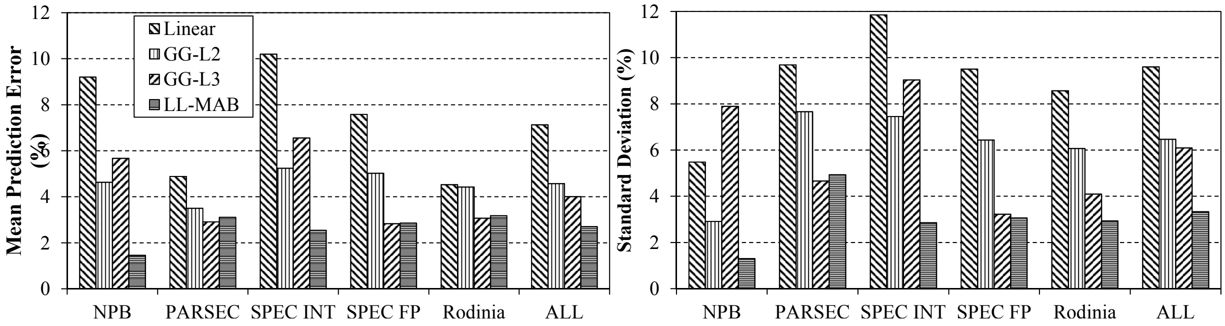
[5] SPEC CPU2006: All 29 benchmarks with *ref* inputs.



(a) Average absolute prediction error.

(b) Standard deviation of the absolute prediction error.

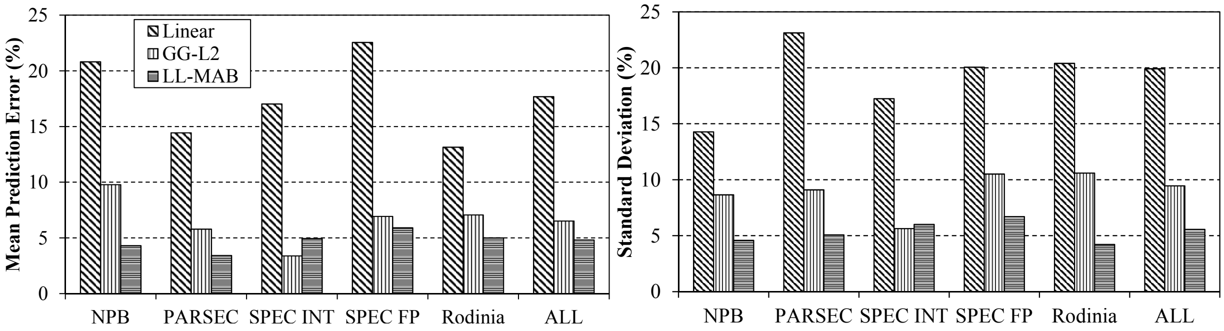
Figure 2: Average and standard deviation of prediction errors on the AMD Phenom^{TM}II 1090T processor.



(a) Average absolute prediction error.

(b) Standard deviation of the absolute prediction error.

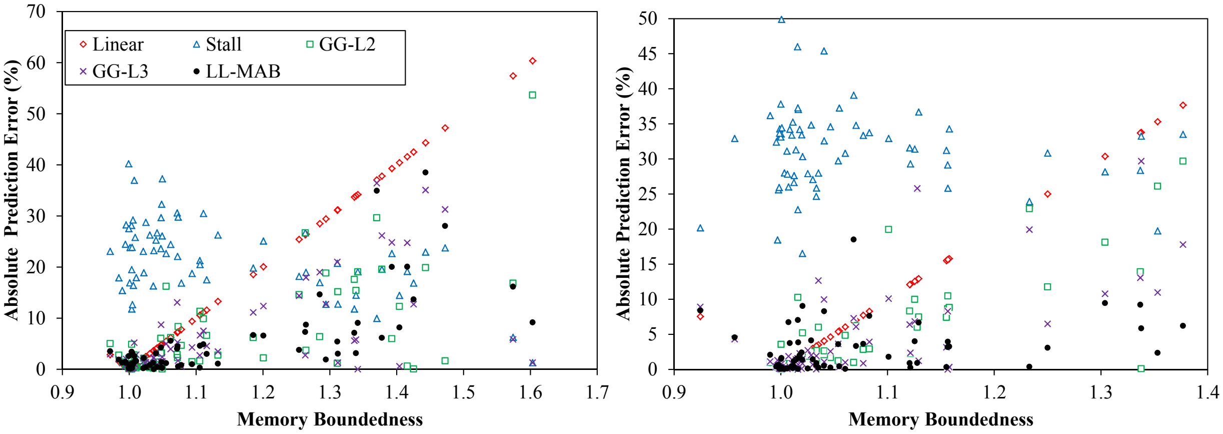
Figure 3: Average and standard deviation of prediction errors on the AMD Opteron^{TM}4386 processor.



(a) Average absolute prediction error.

(b) Standard deviation of the absolute prediction error.

Figure 4: Average and standard deviation of prediction errors on the AMD A10-5800K processor.



(a) AMD Phenom^{TM}II 1090T processor

(b) AMD Opteron^{TM}4386 processor

Figure 5: Prediction error vs. measured memory boundedness (higher means more time in the memory system).

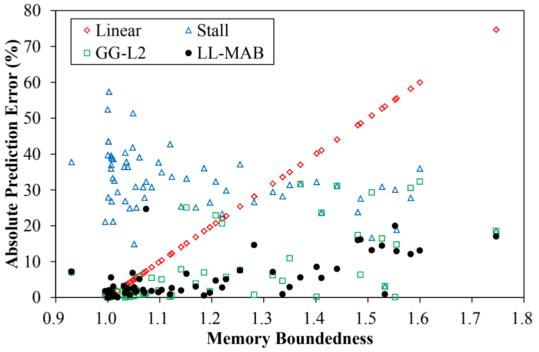


Figure 6: Error vs. memory boundedness on the AMD A10-5800K processor.

3. LL-MAB is more flexible to system configuration changes. For example, changing the DRAM frequency of a machine would not impact LL-MAB. Other models would require retraining.

Unlike the leading loads results shown in the literature, LL-MAB has a higher average error rate, between 2.5% and 5%. All three initial papers that modeled LL showed average error rates of 0.2%, though Miftakhutdinov et al. ran simulations with a more complex memory system that showed worse LL results [9]. Regardless, these results used LL counters that did not have the limitations of our MAB counter. For instance, they only counted misses to the LLC, they had no hardware prefetchers, and (often) assumed a constant delay to memory.

Others (such as those detailed by Rountree et al. [10]) demonstrate regression models on real hardware with better accuracy than LL-MAB. We did not study these in detail, because they have the disadvantage of requiring offline training and more hardware counters.

5 Conclusion and Future Work

In this paper, we presented LL-MAB, the first DVFS performance prediction model based on leading loads implemented on existing hardware. Experiments show it has better prediction accuracy than other state-of-the-art models. Moreover, it requires fewer hardware counters, is easier to use, and has less error variance. Because it is built using existing hardware, it can easily be used by software to enable online DVFS performance prediction with no further hardware changes.

Future work could include using the LL-MAB predictor over short periods for fine-grained DVFS decisions. Similarly, a regression model with this counter may show even better performance than previous regression models. Because LL-MAB requires so few hardware counters, it may also be possible to do online power estimation by monitoring other energy-hungry events.

There are also simple modifications that could increase the accuracy of the MAB event, such as filtering prefetches. Unlike the scheme described by Miftakhutdinov et al. [9], which would require at least an adder for every MAB, these approaches may yield better results with little added hardware.

Acknowledgements

We would like to sincerely thank Barry Rountree for sharing his leading loads expertise and for his help discovering the relationship between leading loads and MAB occupancy. We would also like to thank Vasileios Spiliopoulos, for his help implementing our GG model, and the anonymous reviewers. This work is partially supported by 863 Program of China (2012AA010905), NSFC (61272144, 61272143) and NUDT/Hunan Innov. Fund. For PostGrad. (B120604, CX2012B029).

References

[1] Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Finebert, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. The NAS Parallel Benchmarks. Tech. Rep. RNR-94-007, March 1994.

[2] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Int'l Conf. on Parallel Architectures and Compilation Techniques* (2008).

[3] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *IEEE Int'l Symp. on Workload Characterization* (2009).

[4] Eeckhout, L. *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 2010.

[5] Eyerman, S., and Eeckhout, L. A Counter Architecture for Online DVFS Profitability Estimation. *IEEE Trans. on Computers 59*, 11 (2010), 1576–1583.

[6] Henning, J. L. SPEC CPU2006 Benchmark Descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[7] Keramidas, G., Spiliopoulos, V., and Kaxiras, S. Interval-Based Models for Run-Time DVFS Orchestration in Superscalar Processors. In *Int'l Conf. on Computing Frontiers* (2010).

[8] McVoy, L., and Staelin, C. lmbench: Portable tools for performance analysis. In *USENIX Annual Technical Conf.* (1996).

[9] Miftakhutdinov, R., Ebrahimi, E., and Patt, Y. N. Predicting Performance Impact of DVFS for Realistic Memory Systems. In *Int'l Symp. on Microarchitecture* (2012).

[10] Rountree, B., Lowenthal, D. K., Schulz, M., and de Supinski, B. R. Practical Performance Prediction Under Dynamic Voltage Frequency Scaling. In *Int'l Green Computing Conf. and Workshops* (2011).

[11] Snowdon, D. C., Linden, G. V. D., Petters, S. M., and Heiser, G. Accurate Run-Time Prediction of Performance Degradation Under Frequency Scaling. In *Workshop on Operating System Platforms for Embedded Real-Time Applications* (2007).

[12] Spiliopoulos, V., Kaxiras, S., and Keramidas, G. Green Governors: A Framework for Continuously Adaptive DVFS. In *Int'l Green Computing Conf. and Workshops* (2011).

HaPPy: Hyperthread-aware Power Profiling Dynamically

Yan Zhai
University of Wisconsin
yanzhai@cs.wisc.edu

Xiao Zhang, Stephane Eranian
Google Inc.
{xiaozhang,eranian}@google.com

Lingjia Tang, Jason Mars
University of Michigan
{lingjia,profmars}@eesc.umich.edu

Abstract

Quantifying the power consumption of individual applications co-running on a single server is a critical component for software-based power capping, scheduling, and provisioning techniques in modern datacenters. However, with the proliferation of hyperthreading in the last few generations of server-grade processor designs, the challenge of accurately and dynamically performing this power attribution to individual threads has been significantly exacerbated. Due to the sharing of core-level resources such as functional units, prior techniques are not suitable to attribute the power consumption between hyperthreads sharing a physical core.

In this paper, we present a runtime mechanism that quantifies and attributes power consumption to individual jobs at fine granularity. Specifically, we introduce a hyperthread-aware power model that differentiates between the states when both hardware threads of a core are in use, and when only one thread is in use. By capturing these two different states, we are able to accurately attribute power to each logical CPU in modern servers. We conducted experiments with several Google production workloads on an Intel Sandy Bridge server. Compared to prior hyperthread-oblivious model, HaPPy is substantially more accurate, reducing the prediction error from 20.5% to 7.5% on average and from 31.5% to 9.4% in the worst case.

1 Introduction

As more of the world's computation moves into large-scale datacenter infrastructures, power management and provisioning becomes increasingly important. In fact, prior work [4] shows that the cost of powering the servers housed in these infrastructures comprises about 30% of the total cost of ownership (TCO) of modern datacenter infrastructures. As we are reaching the limits of current power delivery systems, many datacenter infrastructures house more machines than can be powered by the supply infrastructure [17]. In tandem with these trends, datacenter designers and operators have been investigating techniques to manage the available power resources via software techniques such as power-capping [13, 11], scheduling [12], and energy accounting/pricing [28], among others. Software power capping and provisioning techniques ensure that servers do not use more than a specified power threshold by suspending a subset of jobs. Scheduling can also be used to limit processor utilization to reach energy consumption goals. Beyond power budgeting, pricing the power consumed by jobs in datacenters is also important in multi-tenant environments.

One capability that proves critical in enabling software to monitor and manage power resources in large-scale datacenter infrastructures is the attribution of power consumption to the individual applications co-running on a single server. This ability allows software to control power consumption at the level of individual applications on a single machine, as well as across entire clusters. However, accurate attribution on real-world commodity hardware has proven challenging for modern server designs, particularly due to the fact that *simultaneous multi-threading*, (or *hyperthreading* [14]) is now commonplace in current server designs.

Processors that are *hyperthreaded* allow two or more *hardware thread contexts* to share a single physical core. Although the OS views each hardware thread context as a logical CPU, a number of core-level resources are shared across contexts such as functional units, alias register, and cache resources, among others. Modern processors do not provide specific power monitors for each hardware thread context and thus attributing the power consumption of individual processes and threads across logical CPUs has proven particularly challenging.

In this work, we present **HaPPy**, a **H**yperthread-**a**ware **P**ower **P**rofiling D**y**namically. HaPPy is able to dynamically and near instantaneously attribute the power consumed (in watts) to individual processes or threads. To the best of our knowledge, this is the first such hyperthread-aware power estimation approach. Central to HaPPy is an estimation model that uses Intel *Running Average Power Limit* (RAPL) power/performance monitoring interface [14] that is widely available on current commodity servers (Sandy Bridge/Ivy Bridge/etc). Although RAPL provides no power monitoring information of individual cores nor hardware thread contexts, HaPPy uses a novel execution isolation technique implemented on top of existing performance counter tool to predict the power consumed by individual threads.

We evaluate HaPPy on six data-intensive Google production workloads using real commodity server configurations found in datacenters. Compared to prior work, HaPPy is substantially more accurate and reduces the prediction error from 20.5% to 7.5% on average and from 31.5% to 9.4% in worse cases.

2 Background

The primary goal of this work is provide a technique to enable the attribution of power consumption to individual threads. In this section, we first describe the need for power estimation and the most related works. Then, we describe the underlying hardware monitoring interface that underpins our HaPPy approach.

2.1 Need for Power Estimation

Power estimation for individual jobs is critical for power management systems in datacenters [13, 11, 17, 12, 28]. To lower the total cost of ownership, particularly the cost of power infrastructures, modern datacenters are often designed to house more servers than can be powered by the underlying power supply infrastructure. At peak time, the power demand of the datacenter may surpass the supply of the power infrastructure, in which case power capping techniques are applied to lower the demand to under the provisioning threshold. There are various types of power capping techniques, including suspending or limiting the processor utilization of certain jobs. These approaches require accurate power estimation for individual jobs. Power estimation allows us to accurately identify the minimum amount of power-hungry jobs the system needs to suspend given the target power demand threshold. A more conservative power capping system without the power estimation might need to suspend all low-priority jobs, which is much less cost-effective. In addition to power capping, power estimation is also critical for facilitating accurate pricing and accounting in multi-tenant cloud infrastructures. Accurate power usage estimation for individual applications on a shared server allows us to design power-based billing and pricing for cloud infrastructure users.

2.2 State of Power Estimation in Datacenters

Power constraints are well recognized as one of the primary limiting factors for datacenter design, and there is a significant body of work [19, 13, 11, 26] targeting advanced power estimation and management in datacenters. Two works emerge as most related. The work by Fan *et al.* presents power provisioning designs for datacenters [11]. The model presented in this paper focuses on coarse-granularity prediction, which is suitable for its goal. However, it is hyperthread-oblivious and incurs high inaccuracy when directly applied to attributing total server power to individual co-running tasks running on hyperthreaded processors. The work by Shen *et al.* models CPU power at a fine-grained server requests level on hyperthreading disabled servers [26]. Our work is complementary to both of these important contributions as our hyperthread aware CPU power modeling is applicable to tasks concurrently running on a hyperthreaded machines.

2.3 The RAPL Interface

Recently Intel released the RAPL *model specific registers* (MSRs). These performance counters enable software to read processor energy consumption at run time on newer Intel processors such as Sandy Bridge and Ivy Bridge. RAPL MSRs separate processor energy consumption into three parts: `pp0`, `package`, and `dram`. `pp0` counts total energy consumed by all cores of a processor (note that RAPL does not provide per-core measurements on Sandy Bridge or Ivy Bridge); `package` includes both cores and uncore (e.g. last-level-cache) energy consumption; `dram` here means on-chip dram channels, not the commonly referred off-chip memory DIMM. Total processor energy consumption can be calculated by aggregating `package` and `dram` readings. The reported energy during a given time window can then be converted to the average power.

The Linux kernel provides a powerful open-source tool, called *perf* [1], to configure and monitor hardware performance counters. We have extended this interface to enable access to Intel's RAPL counters. The extension is implemented as a separate socket-level performance monitoring unit (PMU). To monitor energy consumption of a multi-socket system, it is only necessary to monitor the RAPL events from one CPU on each socket. Our *perf* patch has been open-sourced [2] and will appear in upstream kernels (Linux 3.14 and newer).

3 Power Modeling

In this section, we first present a hyperthread-oblivious model commonly used in prior work. We then discuss why it is insufficient and inaccurate on modern servers with hyperthreading. Finally, we present our hyperthread-aware model that can accurately attribute power consumption across co-running tasks.

3.1 Hyperthread-oblivious Model

We first present a hyperthread-oblivious (HT-oblivious) model, which is used in prior work for event-driven power accounting [6, 26]. The model is based on the hypothesis that the power consumption of a task is proportional to the amount of computation CPUs perform for that task, and one can estimate the amount of CPU computation using hardware events, such as CPU cycles and instructions.

Figure 1 presents the correlation between applications' power consumption and their aggregated non-halted CPU cycle[1] counts (total cycle counts for all

[1] Non-halted means CPU is not executing the "halt" instruction in x86 instruction set.

workload	description	characteristics
bigtable (BT) [8]	Distributed storage system for managing structured data	Memory-intensive
web-index (IDX) [5]	Web indexing	CPU-intensive
youtube-encoding(YTB)	Youtube video encoding	Floating point-intensive
warp-correction (IMG)	Corrects warped images in scanned material	CPU-intensive
mapreduce (MR) [10]	Map-reduce benchmark written in Sawzall [24] script	Memory-intensive
rpc-bench (RPC)	Google rpc call benchmark	CPU-intensive

Table 1: *Brief description of Google's internal applications used in this study. All applications are memory resident and fully utilize server memory and CPUs.*

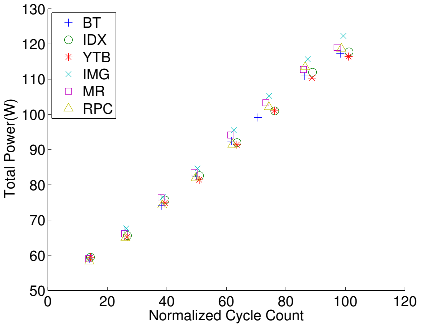


Figure 1: *Correlation of power with non-halted CPU cycle.*

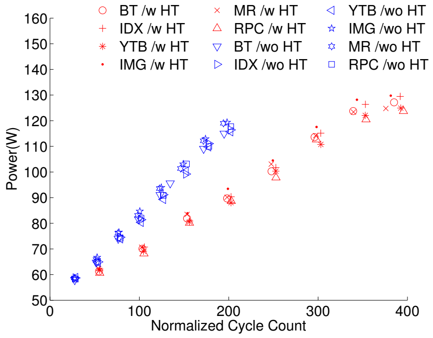


Figure 2: *Correlation between power and cycle when hyperthreads are enabled.*

threads of an application). In these experiments, we use several diverse real Google workloads (see description in table 1) and an Intel Sandy Bridge server with the same configuration found in production. We run N instances of each application on N physical cores on a server. We do not use hyperthreading in this experiment, so only 1 hyperthread of a physical core is used. During the experiment, we collect the total processor power consumption and aggregated non-halted CPU cycles using *perf* [1]. As demonstrated in Figure 1, the aggregated CPU cycles are strongly correlated with the power consumption (linear correlation coefficient 0.99).

Besides non-halted CPU cycles, we also examined other metrics including instruction count, last-level-cache reference and miss, through a wide range of microbenchmarks, including a busy-loop benchmark (high instruction issue rate), a pointer chasing benchmark (high cache miss rate), a CPU and memory intensive benchmark (to mimic power virus behavior), and a set of bubble-up benchmarks that incur adjustable amounts of pressure on the memory systems [27]. Our conclusion is that non-halted cycle is the best to correlate power (linear correlation coefficient above 0.95). This finding is consistent with prior work [11] which suggests a strong correlation between the machine-level power consumption and CPU utilization.

Based on the correlation between the power consumption and the cycle count when hyperthreading is not in use, the power consumption across all currently running tasks can be attributed simply based on each task's cycle count. This HT-oblivious model for attributing power is as follows:

$$Power(Task_i) = Total\_Active\_Power \times \frac{Cycle(Task_i)}{\sum_{j=1}^{m} Cycle(Task_j)} \qquad (1)$$

3.2 Why is accounting hyperthreading important?

The HT-oblivious model assumes that the power consumption of a task is strongly correlated with the aggregated CPU cycles of the task. However, as we will demonstrate in this section, this is no longer the case when hyperthreads are used. Figure 2 presents the correlation between the measured power consumption and the aggregated cycle counts when tasks use hyperthreads (2 tasks pinned to the two hyperthreads of each physical core) versus when tasks do not use hyperthreads (only 1 task pinned to each physical core). As presented in Figure 2, *the aggregated CPU cycles of a task are not strongly correlated with its power consumption when hyperthreading may be in use*. For example, as shown in the figure, when the normalized cycle count is around 200, the power consumption can be wildly different ranging between 85w and 120w, almost 40% difference.

To illustrate the reason behind this 40% discrepancy, imagine when both hyperthreads (HT) of a physical core are in use, the aggregated CPU cycles may double comparing to the scenario when only 1 HT per core is in use (each HT is a logical CPU). However, the power consumption is not doubled. Actually, there is only a slight power increase over the scenario when 1 HT/core is used. This is because that two hyperthreads of a physical core share many low-level hardware resources (such as functional units), thus only incur slight power increase when both are active. *A basic hyperthreading-oblivious model does not distinguish these two scenarios (with and without hyperthreading), and therefore is inaccurate*.

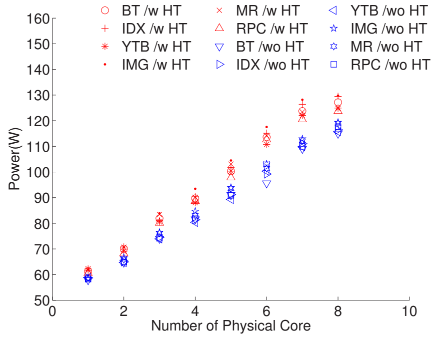


Figure 3: *Power comparison when using only one or both hyperthreads of a physical core.*

Figure 3 further demonstrates the power consumption difference between using no hyperthreading (1 task pinned to only one logic CPU of a physical core) and using hyperthreading (2 tasks pinned to the two hyperthreads, logical CPUs, of a physical core). The x-axis shows the number of physical cores used in each experiment. For n physical cores, we execute n replica of a task for w/o hyperthreading scenario and $2n$ replica of the task for w/ hyperthreading scenario. The y-axis shows the measured total CPU power. Again, as presented in Figure 3, *accounting hyperthreading is critical for the accuracy of a power model*. For example, when attributing power for 8 (single-threaded) tasks, it is important to differentiate whether 8 tasks are running on eight cores (∼115w total and 14 w/task) or on four physical cores with 2 hyperthreads each core (∼90w total and 11 w/task, 30% less than 14 w/task) or the mix of both scenarios. The Evaluation section (Section 4) will further demonstrate the inaccuracy when one fails to acknowledge the hyperthread configurations. Also note that, as shown in Figure 3, the ratio between power consumption when both hyperthreads are in use and that when only one hyperthread is in use is about 1.1. We refer this ratio as R_{ht} for the rest of this paper.

3.3 Hyperthread-aware Model

In this section, we present a novel hyperthread-aware (HT-aware) model that addresses the challenge of accounting per task power consumption when tasks may use hyperthreads. We first break down the total CPU power consumption into *static power* and *active power* , and then focus on modeling the active power. To attribute the active power across all co-running tasks, our model first attributes the power consumption for each physical core using a novel technique to account for how tasks are taking advantage of hyperthreading of the core. We then attribute the power consumption of each task based on the cycles each task executes on each core.

3.3.1 Attributing active power among physical cores

Static Power - Processors often consume a small amount of power just to be active, even when there is not much computation activity. For example, linearly extrapolating the data points in Figure 3 shows that when 0 core is in use, the power consumption is around 50 watt. This means that there is around 50 watt of power consumption even when there is minimum core activity. We refer to this power consumption as the *static power* and use linear extrapolation to estimate it. The static power on our test machine is estimated to be 52.5 watts.

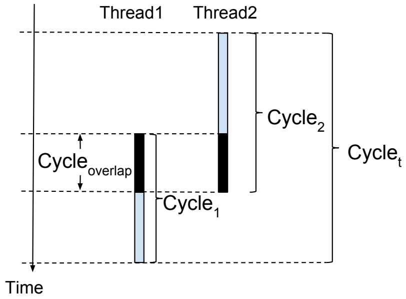


Figure 4: *Illustration of how we capture the detailed utilization of a physical core using three counters. Thread 1 and 2 are sibling hyperthreads of a physical core, and their non-halted CPU cycles are respectively $Cycle_1$ and $Cycle_2$. $Cycle_t$ represents non-halted cycles when* at least *one of the two hyperthreads of a physical core is active.*

Attributing the active power to each physical core - We calculate the active power using the total CPU power consumption minus the static power estimated as discussed above. To attribute the active power across physical cores, our model takes advantage of three hardware counters. As illustrated in Figure 4, $Cycle_1$and $Cycle_2$ are non-halt CPU cycles respectively for thread 1 and 2. $Cycle_t$ is CPU cycles when *at least* one of the two hyperthreads of a physical core is running [2].

From these three counters, we can infer:

$$Cycle_{overlap} = Cycle_1 + Cycle_2 - Cycle_t \qquad (2)$$

$$Cycle_{nonoverlap} = Cycle_t - Cycle_{overlap} \qquad (3)$$

$Cycle_{overlap}$ is the portion of time when both hyperthreads of a physical core are running, while

[2] In *perf* tool, $Cycle_1$ and $Cycle_2$ are obtained by cpu event 0x3c with umask=0x00, while $Cycle_t$ can be obtained by same event with umask=0x00 and any=1.

$Cycle_{nonoverlap}$ is that when only one thread is running. Using $Cycle_{overlap}$ and $Cycle_{nonoverlap}$, we define the weighted cycle of a physical core as follows:

$$Cycle_{weighted}(Core) = R_{ht} \times Cycle_{overlap} + Cycle_{nonoverlap} \quad (4)$$

The **intuition** here is that *when two hyperthreads are both executing on a physical core, the power consumption of the physical core is R_{ht} times its power consumption when only one hyperthread is executing on the core.* R_{ht} is computed from the data in Figure 3 to be 1.1. So by calculating the time period when two hyperthreads are executing ($Cycle_{overlap}$) and that when only one is executing ($Cycle_{nonoverlap}$), we can calculate a weighted core utilization ($Cycle_{weighted}$) and use it to estimate each core's power consumption.

Now we can attribute the total active power of a processor among individual cores proportional to each core's hyperthread-weighted $Cycle_{weighted}$:

$$Active\_Power(Core_i) = Total\_Active\_Power \times \frac{Cycle_{weighted}(Core_i)}{\sum_{j=1}^{n} Cycle_{weighted}(Core_j)} \quad (5)$$

3.3.2 Attributing active core power to hyperthreads

Following the same principle of Equation 4, we can calculate the weighted cycles for individual hyperthreads on each core:

$$Cycle_{weighted}(HT_i) = R_{ht} \times \frac{Cycle_{overlap}}{2} + (Cycle_i - Cycle_{overlap}) \quad (6)$$

HT_i is one of two hyperthreads of a physical core. Recall that $Cycle_{overlap}$ indicates the time when both threads are running (Equation 2), in which case we attribute the power to each individual hyperthread evenly. $Cycle_i - Cycle_{overlap}$ represents the time thread i runs alone, in which case the thread is attributed the total power consumed by the core. Using $Cycle_{weighted}(HT_i)$ calculated by Equation 6 and $Active\_Power(Core_i)$ calculated by Equation 5, we can proportionally attribute the total active power of a core to each hyperthread on that core using the following equation:

$$Active\_Power(HT_i) = Active\_Power(Core) \times \frac{Cycle_{weighted}(HT_i)}{Cycle_{weighted}(Core)} \quad (7)$$

3.4 Mapping from hardware to applications

Reserving logical CPUs is a common practice in datacenters to achieve better performance isolation [22]. With this technique, the process threads associated with a job run on dedicated CPUs using containers and the `set_affinity` API [3]. Our approach attributes active power to such jobs by calculating the power consumption of the hyperthread contexts associated with each hosted process as shown in Equation 7. Reserving CPUs is often used for minimizing performance interference to latency-critical jobs as well as in Infrastructure as a Service (IaaS) type of multi-tenant cloud services. We use this execution approach as the basis of our evaluation.

When CPUs are time-shared by multiple jobs, our models can be used to capture the power consumption change at each context switch using Equation 7. The cost of reading the performance counters is typically hundreds of nanoseconds while the remaining cost of a context switch is on the order of microseconds [20]. It is important to note that the OS scheduler only needs to save threads' co-run performance counter information at every context switch, more complex calculations can be deferred to a coarser scale (seconds) or on demand.

4 Evaluation

In this section, we evaluate the accuracy of our model using production Google applications listed in Table 1. We also duplicate our experiment using SPEC benchmarks for repeatability. Our Google configuration is a 2.6GHz Intel Sandy Bridge machine, equipped with 2 processor sockets. Each socket has 8 cores, each with two hyperthreads. Only one socket is used in our experiments. The testbed runs a customized Linux kernel with necessary RAPL support. We collect energy readings via *perf* tool every 10 seconds and report the average power during a 300-second application execution.

4.1 Methodology

In our experimental setup, we co-run two jobs on a processor socket and pin them to disjoint sets of cores. The first job spawns $2N$ processes on N cores, while the second job spawns N processes on another N cores. We scale up N from 1 to 4 in our experiments. With this configuration, the first job makes full usage of all hyperthreads ($2N$ hyperthreads on N cores), while the second job uses half of the available hyperthreads (N hyperthreads on N cores). We first calculate total active power consumed by both jobs as $Power_{total}$ (measured total processor power minus static power). We then estimate each job's power consumption, $Power_1$ and $Power_2$, using both HT-oblivious model (Equation 1) and our HT-aware model (Equations 2 - 7).

To evaluate the accuracy for estimating per job power consumption, we remove one job from the server and measure the power consumption of the remaining job as $Power'_{total}$. The delta between $Power_{total}$ and $Power'_{total}$ is the actual active power of the removed job. We refer to this measured per task power as "Oracle", and use it as an evaluation baseline in the following section.

Workload	HT-oblivious		HT-aware	
	Avg Error	Max Error	Avg Error	Max Error
{bigtable (BT), warp-correction (IMG)}	5.8w(23.1%)	11.9w(28.5%)	1.8w(7.0%)	3.7w(8.6%)
{web-index (IDX), mapreduce (MR)}	5.1w(19.4%)	13.4w(31.0%)	2.4w(9.3%)	4.4w(10.2%)
{youtube-encoding(YTB), rpc-bench (RPC)}	4.8w(19.1%)	13.9w(34.5%)	1.5w(6.2%)	3.9w(9.6%)
Average	5.2w(20.5%)	13.1w(31.3%)	1.9w(7.5%)	4.0w(9.4%)

Table 2: *Average and maximal errors for Google benchmarks, both in absolute watt and relative percentage, of two models when using Oracle as baseline. Error in percent is calculated as* $\frac{|Oracle_{avg}-Model|}{Oracle_{avg}}$ *and* $\frac{|Oracle_{max}-Model|}{Oracle_{max}}$.

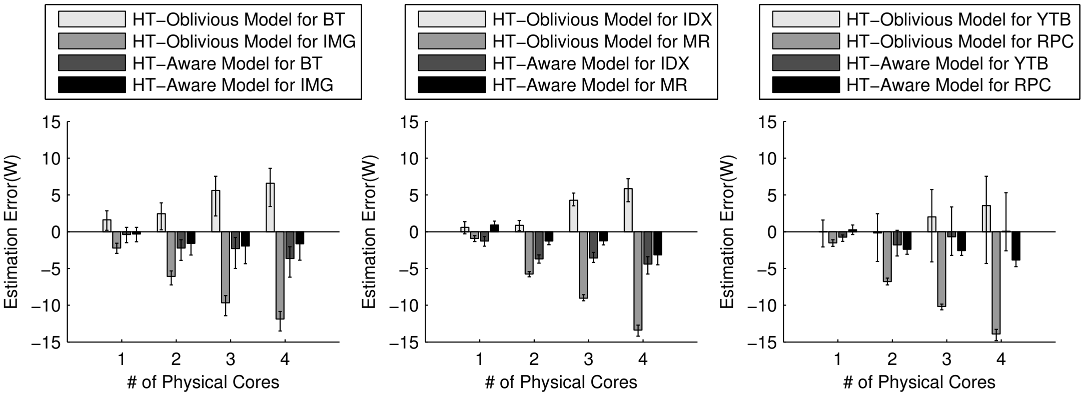


Figure 5: *Results for three sets of workload {bigtable (BT), warp-correction (IMG)}, {web-index (IDX), mapreduce (MR)}, and {youtube-encoding(YTB), rpc-bench (RPC)}*

4.2 Results

We conduct the evaluation using three pairs of co-running applications, chosen arbitrarily: {bigtable (BT), warp-correction (IMG)}, {web-index (IDX), mapreduce (MR)}, {youtube-encoding(YTB), rpc-bench (RPC)} (applications are described in Table 1). As discussed in Section 4.1, for each pair, we conduct four experiments varying the number of physical cores *N* from 1 to 4. The first job in a pair spawns 2*N* processes running on *N* cores, while the second only spawns *N* processes on another *N* cores. Each experiment runs three times. Both the average values and standard deviations are reported.

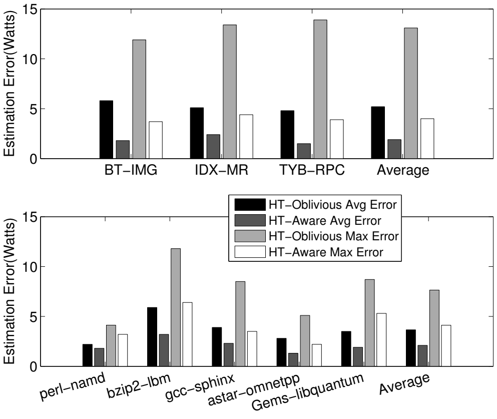


Figure 6: *Improvement of power estimation for both Google and SPECPU benchmarks.*

Figure 5 presents our experimental results. It shows the power attributing results for each co-run pair. As shown in the figure, the prediction accuracy of our HT-aware model outperforms the HT-oblivious model. Figure 5 shows that the HT-oblivious model tends to overestimate jobs with both hyperthreads running (*i.e.*, bigtable (BT), web-index (IDX), and youtube-encoding(YTB)) and underestimate jobs with only one hyperthread running (*i.e.*, warp-correction (IMG), mapreduce (MR), and rpc-bench (RPC)). These prediction errors are expected since the HT-oblivious model solely depends on CPU cycles. In contrast, our HT-aware model takes architecture details into consideration and is more accurate in all cases. As summarized in Table 2, it on average reduces error from 20.5% to 7.5% when compared to HT-oblivious. The maximal error of HT-aware prediction is significantly less than the HT-oblivious model, reducing from ∼13 watts error (or 31.3%) for the HT-oblivious model to ∼4 watts (or 9.4%) for our HT-aware model.

4.3 SPEC results

To demonstrate the repeatability of our experiments beyond Google applications, we duplicated the experiments using SPEC CPU2006 benchmarks on another Sandy Bridge machine. On this machine, each CPU socket has six 1.9GHz physical cores. In these experiments, we used 10 SPEC benchmarks and randomly group them in 5 pairs. Figure 6 presents the power prediction error achieved by HaPPy versus the hyperthread-oblivious model. As shown in the figure, there is a significant improvement in prediction accuracy for both Google and SPEC workloads.

5 Related Work

Several techniques have been proposed to predict the server power [6, 9, 18, 15, 21, 23]. For example, Bellosa proposed an event driven approach for power modeling. Choi *et al.* discussed power prediction and capping in consolidated environment [9]. Lee *et al.* designed a regression model for power prediction in hardware simulators [18], whereas our model is applicable for real machines. Isci *et al.* presented a framework to collect and analyze power phases [15]. These work either do not explicitly address hyperthreaded servers or simply disable hyperthreading. Fine-grained power profiling tools are also proposed [25]. Shen *et al.* designed a power container to profile server request level power [26]. Kansal *et al.* and Bertran *et al.* used system events to model application level power [16, 7]. Again these models are not aware of hyperthreads. Power management in data center has attracted much research attention recently [19, 13, 11]. These power management techniques require accurate power estimation.

6 Conclusion

In this paper, we present a simple and accurate hyperthread-aware power model to attribute power consumption of a server to individual co-running applications. By leveraging on-chip energy counters and *perf* tool, we prototype our model as a lightweight runtime task power profiler. Our evaluation using Google commercial benchmarks shows that the prediction accuracy of our model is significantly better than the state-of-the-art hyperthread-oblivious model.

Acknowledgement

We would like to thank Keith Smith for his valuable feedback. This work was partially supported by NSF Awards CCF-SHF-1302682, CNS-CSR-1321047 and a Google Research Award.

References

[1] `https://perf.wiki.kernel.org/`.

[2] `https://lkml.org/lkml/2013/10/7/359`.

[3] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 1999.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool Publishers, 2nd edition, 2013.

[5] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, Mar. 2003.

[6] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proc. of the SIGOPS European Workshop*, Kolding, Denmark, Sept 2000.

[7] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and responsive power models for multicore processors using performance counters. In *Proc. of the 24th ACM International Conference on Supercomputing (SC)*, pages 147–158, 2010.

[8] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of the 7th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 205–218, 2006.

[9] J. Choi, S. Govindan, B. Urgaonkar, and A. Sivasubramaniam. Profiling, prediction, and capping of power consumption in consolidated environments. In *Proc. of Modeling, Analysis and Simulation of Computers and Telecommunication Systems (MASCOTS)*, pages 1–10, 2008.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of the 6th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2004.

[11] X. Fan, W. D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proc. of the 34th annual International Symposium on Computer Architecture (ISCA)*, pages 13–23, 2007.

[12] I. Goiri, W. Katsak, K. Le, T. D. Nguyen, and R. Bianchini. Parasol and greenswitch: Managing datacenters powered by renewable energy. In *Proc. of 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 51–64, 2013.

[13] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proc. of the 4th ACM European Conference on Computer systems (EuroSys)*, pages 317–330, 2009.

[14] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, volume 3: System programming guide, 2013.

[15] C. Isci and M. Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *Proc. of 12th Int'l Symp. on High Performance Computer Architecture (HPCA)*, pages 121–132, 2006.

[16] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. *ACM SIGMETRICS Performance Evaluation Review*, 36(2):26–31, 2008.

[17] V. Kontorinis, L. E. Zhang, B. Aksanli, J. Sampson, H. Homayoun, E. Pettis, D. M. Tullsen, and T. S. Rosing. Managing distributed UPS energy for effective power capping in data centers. In *Proc. of the 39th annual International Symposium on Computer Architecture (ISCA)*, pages 488–499. IEEE, 2012.

[18] B. C. Lee and D. M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 185–194. ACM, 2006.

[19] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. In *Proc. of the 4th International Conference on Autonomic Computing (ICAC)*, 2007.

[20] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proc. of the Workshop on Experimental Computer Science (ExpCS)*, 2007.

[21] T. Li and L. K. John. Run-time modeling and estimation of operating system power consumption. In *Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 160–171, 2003.

[22] 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 *Proc. of the 44th annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, New York, NY, USA, 2011. ACM.

[23] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta. Evaluating the effectiveness of model-based power characterization. In *Proc. of the USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[24] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 13(4):277–298, Oct. 2005.

[25] A. Roy, S. M. Rumble, R. Stutsman, P. Levis, D. Mazières, and N. Zeldovich. Energy management in mobile devices with the Cinder operating system. In *Proc. of the 6th ACM European Conference on Computer systems (EuroSys)*, pages 139–152, 2011.

[26] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and C. Zhuan. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *Proc. of 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, Texas, Mar. 2013.

[27] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-Flux: Precise online QoS management for increased utilization in warehouse scale computers. In *Proc. of the 40th annual International Symposium on Computer Architecture (ISCA)*, 2013.

[28] Q. Zheng and B. Veeravalli. Utilization-based pricing for power management and profit optimization in data centers. *Journal of Parallel and Distributed Computing*, 72(1):27–34, 2012.

Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks

Ran Liu ‡ † , Heng Zhang †, Haibo Chen †

‡*Software School, Fudan University*
†*Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University*

ABSTRACT

Reader-writer locks (rwlocks) aim to maximize parallelism among readers, but many existing rwlocks either cause readers to contend, or significantly extend writer latency, or both. Further, some scalable rwlocks cannot cope with OS semantics like sleeping inside critical sections, preemption and conditional wait. Though truly scalable rwlocks exist, some of them cannot handle preemption, sleeping inside critical sections, or other important functions required inside OS kernels. This paper describes a new rwlock called the passive reader-writer lock (prwlock) that provides scalable read-side performance as well as small writer latency for TSO architectures. The key of prwlock is a version-based consensus protocol between multiple non-communicating readers and a pending writer. Prwlock leverages *bounded staleness* of memory consistency to avoid atomic instructions and memory barriers in readers' common paths, and uses message-passing (e.g., IPI) for straggling readers so that writer lock acquisition latency can be bounded. Evaluation on a 64-core machine shows that prwlock significantly boosts the performance of the Linux virtual memory subsystem, a concurrent hashtable and an in-memory database.

1 INTRODUCTION

Reader-writer locking is an important synchronization primitive that allows multiple threads with read accesses to a shared object when there is no writer, and blocks all readers when there is an inflight writer [13]. While ideally rwlock should provide scalable performance when there are infrequent writers, it is widely recognized that traditional centralized rwlocks have poor scalability [9, 25, 10]. For example, it is explicitly recommended to not use rwlocks unless readers hold their locks for a sufficiently long time [9].

While there have been a number of efforts to to improve the scalability of rwlocks, prior approaches either require memory barriers and atomic instructions in readers [22, 18], or significantly extend writer latency [5], or both [12, 2]. Further, many prior designs cannot cope with OS semantics like sleeping inside critical section, preemption and supporting condition synchronization (e.g., wait/signal) [12, 2]. Hence, researchers sometimes relax semantic guarantees by allowing readers to see stale data (i.e., RCU [21]). While RCU has been widely used in Linux kernel for some relatively simple data structures, it, however, would require non-trivial effort for some complex kernel data structures and may be incompatible with some existing kernel designs [10, 11]. Hence, there are still thousands of usages or rwlocks inside Linux kernel [20].

This paper describes the prwlock, a scalable rwlock design for read-mostly synchronization for TSO (Total Store Ordering) architectures. Like prior designs such as brlock [12, 2], instead of letting readers actively maintain status regarding inflight readers, prwlock decentralizes such information to each reader and only makes a consensus among readers when a writer explicitly enquires. By leveraging the ordered store property of TSO architectures, such as x86 and x86-64, Prwlock achieves truly scalable reader performance. On TSO, it not only requires no atomic instructions or memory barriers on the common path, but it also limits writer latency when there are concurrent readers.

The key of prwlock is a version-based consensus protocol between multiple non-communicating readers and a pending writer. A writer advances the lock version and waits other readers to see this version to ensure that they have left their read-side critical sections. Unlike prior designs such as brlocks, this design is based on our observation that *even without explicit memory barriers, most readers are still able to see a most-recent update of the lock version from the writer within a small number of cycles*. We call this property *bounded staleness*. For straggling readers not seeing and reporting the version update, prwlock uses a message-based mechanism based on inter-processor interrupts (IPIs) to explicitly achieve consensus. Upon receiving the message, a reader will report to the writer whether it has left the critical section. As currently message passing among cores using IPIs is not prohibitively high [4] and only very few straggling readers require message-based consensus, a writer only needs to wait shortly to proceed.

As a reader might sleep in the read-side critical section, it may not be able to receive messages from the writer. Hence, a sleeping reader might infinitely delay a writer. To address this issue, prwlock falls back to a shared counter to count sleeping readers. As sleeping in read-side critical sections is usually rare, the counter is rarely used and contention on the shared counter will not be a performance bottleneck even if there are a small number of sleeping readers.

Prwlock is built with a parallel wakeup mechanism to improve performance when there are multiple sleeping readers waiting for an outstanding writer. As traditional wakeup mechanisms (like Linux) usually use a shared queue for multiple sleeping readers, a writer needs to wake up multiple readers sequentially, which becomes a scalability bottleneck with the increasing number of readers. Based on the observation that multiple readers can be woken up in parallel with no priority violation in many cases, prwlock introduces a parallel wakeup mechanism such that each reader is woken up by the core where it slept from.

We have implemented prwlock as a kernel mechanism for Linux, which compromises around 300 lines of code (LoC). To further benefit user-level code, we also created a user-level prwlock library (comprising about 500 LoC) and added it to a user-level RCU library (about 100 LoC changes). Prwlock can be used in the complex Linux virtual memory system (which currently uses rwlock), with only around 30 LoC changes. The implementation is stable enough and has passed the Linux Test Project [1]. We have also applied prwlock by substituting for a rwlock in the Kyoto Cabinet database [17].

Performance evaluation on a 64-core AMD machine shows that prwlock has extremely good performance scalability for read-mostly workloads and still good performance when there are quite a few writers. The performance speedup of prwlock over stock Linux is 2.85X, 1.55X and 1.20X for three benchmarks on 64 cores and prwlock performs closely to a recent effort in using RCU to scale Linux virtual memory [10]. Evaluation using micro-benchmarks and the in-memory database shows that prwlock consistently outperforms rwlock in Linux (by 7.37X for the Kyoto Cabinet database).

2 Background and Related Work

2.1 Reader/Writer Lock

The reader/writer problem was described by Courtois et al. [13] and has been intensively studied afterwards. However, most prior rwlocks require sharing states among readers and thus may result in poor critical section efficiency on multicore. Hence, there have been intense efforts to improve rwlocks. Table 1 shows a comparative study of different designs, using a set of criteria related to performance and functionality. The first three rows list the criteria critical to reader performance, including memory barriers, atomic instructions and communication among readers. The next four rows depict whether each design can support sleeping inside critical section (which also implies preemption) and condition wait (e.g., wait until a specific event such as queue is not empty), and whether the lock is writer or reader preference. The last two rows indicate whether the writer in each design has short writer latency when there are a small number of threads, and whether the design retains the original semantics of rwlock.

	Traditional	brlock1	brlock2	C-SNZI	Cohort	RMLock	PRW	Percpu-rwlock	RCU
No memory barrier in read						✓	✓	✓	✓
No atomic instruction in read			✓			✓	✓	✓	✓
No comm. among readers		✓	✓			✓	✓	✓	✓
Sleep inside critical section	✓			✓	✓	✓	✓	✓	✓
Condition wait	✓			✓		✓	✓	✓	-
Writer preference	✓		✓	✓	✓	✓	✓	✓	-
Reader preference	✓				✓				-
Short writer latency w/ small #thread	✓			✓	✓		✓	*	-
Unchanged rwlock semantic	✓	✓	✓	✓	✓		✓	✓	

*The writer latency of Percpu-rwlock is extremely long in most cases

Table 1: A comparison of synchronization primitives.

Big-reader Lock (brlock): The key design of brlock is trading write throughput for read throughput. There are two implementations of brlock: 1) requiring each thread to obtain a private mutex to acquire the lock in read mode and to obtain all private mutexes to acquire the lock in write mode (brlock1); 2) using an array of reader flags shared by readers and writer (brlock2). However, brlock1 requires heavyweight operations for both reader and writer sections, as the cost of acquiring a mutex is still non-trivial and the cost for the writer is high for a relatively large number of cores (i.e., readers).

Brlock2, like prwlock, uses per-core reader status and forces writers to check each reader's status, and thus avoids atomic instructions in reader side. However, it still requires memory barriers inside inside readers' common paths. Further, both do not support sleeping inside read-side critical sections as there is no centralized writer condition to sleep on and wake up. Finally, they are vulnerable to deadlock when a thread is preempted and migrated to another core. As a result, brlocks are most often used with preemption disabled.

Prwlock can be viewed as a type of brlock. However, it uses a version-based consensus protocol instead of a single flag to avoid memory barriers in readers' common paths and to shorten writer latency. Further, by leveraging a hybrid design, prwlock can cope with complex semantics like sleeping and preemption, making it viable to be used in complex systems like virtual memory.

C-SNZI: Lev et al. [18] use scalable nonzero indicator (SNZI) [16] to implement rwlocks. The key idea is instead of knowing exactly how many readers are in progress, the writer only needs to know whether there are any inflight readers. This, however, still requires actively maintaining reader status in a tree and thus may have scalability issues under a relatively large number of cores [8] due to the shared tree among readers.

Cohort Lock: Irina et al. leverage the lock cohorting [15] technique to implement several NUMA-friendly rwlocks, in which writers tend to pass the lock to another writer within a NUMA node. While writers benefit from better NUMA locality, its readers are implemented using per-node shared counters and thus still suffer from cache contention and atomic instructions. Prwlock is orthogonal to this design and can be plugged into it as a read indicator without memory barriers in reader side.

Percpu-rwlock: Linux community is redesigning a new rwlock, called percpu rwlock [5]. Although, like prwlock, it avoids unnecessary atomic instructions and memory barriers, its writer requires RCU-based quiescence detection and can only be granted after at least one grace period, where all cores have done a mode/context switch. Hence, according to our evaluation (section 6), it performs poorly when there are a few writers, and thus can only be used in the case of having extremely rare writers.

Read-Mostly Lock: From version 7.0, the FreeBSD kernel includes a new rwlock named reader-mostly lock (rmlock). Its readers enqueue special tracker structures into per-cpu queues. A writer lock is acquired by instructing all cores to move local tracker structures to a centralized queue via IPI, then waiting for all the corresponding readers to exit. Like prwlock, it eliminates memory barriers in reader fast paths. Yet, its reader fast path is much longer compared to prwlock, resulting in inferior reader throughput. Moreover, as IPIs need always to be broadcasted to all cores, and ongoing readers may contented on the shard queue, its writer lock acquisition is heavyweight (section 6.2.4). In contrast, prwlock leverages bounded staleness of memory consistency to avoid IPIs in the common case.

2.2 Read-Copy Update

RCU increases concurrency by relaxing the semantics of locking. Writers are still serialized using a mutex lock, but readers can proceed without any lock. As a result, readers may see stale data. RCU delays freeing memory until there is no reader referencing to the object, by using scheduler-based or epoch-based quiescence detection that leverage context or mode switches. In contrast, the quiescence detection (or consensus) mechanism in prwlock does not rely on context or mode switches and is thus faster due to its proactive nature.

RCU's relaxed semantics essentially break the all-or-nothing atomicity in reading and writing a shared object. Hence, it also places several constraints on the data structures, including single-pointer update and readers can only observe a pointer once (i.e., non-repeatable read). This constrains data structure design and complicates programming, since programmers must handle races and stale data and cannot always rely on cross-data-structure invariants. For example, a recent effort in applying RCU to page fault handling shows that several subtle races need to be handled manually [10], which make it very complex and resource-intensive [11]. In contrast, though prwlock can degrade scalability by preventing readers from proceeding concurrently with a single writer, it still preserves the clear semantics of rwlocks. Hence, it is trivial to completely integrate it into complex subsystems, such as address space management.

2.3 Prwlock's Position

As prwlock strives to achieve scalable reader performance with low reader-side latency, it is designed with a simple yet fast reader fast path, which eliminates the need of reader-shared state and even memory barriers. Yet by leveraging bounded staleness for common cases and IPIs for rare cases, its writer latency is still bounded, especially when readers are frequent.

Prwlock targets the territory of RCU where extremely low reader latency is preferred. Compared to RCU, it trades obstruction-free reader access for a much stronger and clearer semantic and much shorter writer latency. Hence, it can be used to improve performance with trivial effort for cases where RCU is hard to apply.

3 Design of prwlock

3.1 Design Rationale

The essential design goal of reader-writer lock (rwlock) is that readers should proceed concurrently, and thus should not share anything with each other. Hence, a scalable rwlock design should require no shared state among readers and no explicit or implicit memory barriers when there are no writers pending. However, typical rwlocks rely on atomic instructions to coordinate readers and writers. On many processors, an atomic instruction implies a memory barrier, which prevents reordering of memory operations across critical section boundary. In this way, readers are guaranteed to see the newest version of data written by the last writer. However, such memory barriers are unnecessary when no writer is present, as there are no memory ordering dependency among readers. Such unnecessary memory barriers may cause significant overhead for short reader critical sections.

Message passing is not prohibitively expensive: Commodity multicore processors resemble distributed systems [4] in that each core has its own memory hierarchy. Each core communicates with others using message passing in essence, but hardware designers add an abstraction (i.e., cache coherence) to emulate a shared memory interface. Such an abstraction usually comes

	IPI Latency (Cycles)	StdDev
AMD 64Core (Opteron 6274 * 4)	1316.3	171.4
Intel 40Core (Xeon E7-4850 *4)	1447.3	205.8

Table 2: IPI latency in different machines

at a cost: due to serialization of coherence messages, sharing contended cache lines is usually costly (up to 4,000 cycles for a cache line read on a 48-core machine [6, 7]) and sometimes the cost significantly exceeds explicit message passing like inter-processor interrupts (IPIs). Table 2 illustrates the pairwise IPI latency on 2 recent large SMP systems, which is 1,316 and 1,447 cycles accordingly. This latency is low enough to be used in rwlocks, whose writer latency usually exceeds several tens of thousands of cycles.

Further, delivering multiple IPIs to different cores can be parallelized so that the cost of parallel IPI is "indistinguishable" from point-to-point interrupt [23]. This may be because point-to-point cache line movement may involve multiple cores depending on the cache line state, while an IPI is a simple point-to-point message.

Bounded staleness without memory barriers: In an rwlock, a writer needs to achieve consensus among all its readers to acquire the lock. Hence, a writer must let all readers see its current status in order to proceed. Typical rwlocks either use an explicit memory barrier or wait for a barrier [5] to make sure the version updates in the reader/writer are visible to each other in order. However, we argue that these are too pessimistic in either requiring costly memory barriers that limit read-side scalability or in significantly extending the writer latency (e.g., waiting for a grace period).

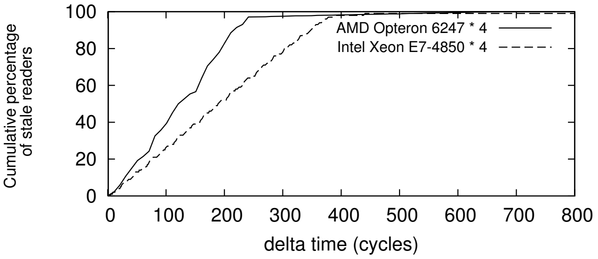


Figure 1: Cumulative percentage of stale readers

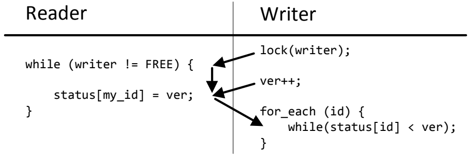
We observe that in commodity processors such as x86(-64), multiple memory updates can usually be visible to other cores in a very short time. We use a microbenchmark to repeatedly write a memory location and read the location on another core after a random delay. We then collect the intervals of readers that see the stale value. Figure 1 shows the cumulative percentage of stale readers along with time; most readers can see the writer's update in a very short time (i.e., less than 400 cycles). This is because a processor will actively flush its store buffer due to its limited size. It is reasonable to simply wait a small amount of time until a reader sees the updated version for the common case, while using a slightly heavyweight mechanism to guarantee correctness.

Memory barrier not essential for mutual exclusion: To reduce processor pipeline stalls caused by memory accesses or other time-consuming operations, modern processors execute instructions out of order and incorporate a store buffer to allow the processor to continually execute after write cache misses. This leads to weaker memory consistency. To achieve correct mutual exclusion, expensive synchronization mechanisms like memory barriers are often used to serialize the pipeline and flush the store buffer.. This may cause notable performance overhead for short critical sections.

Attiya et al. proved that it is impossible to build an algorithm that satisfies mutual exclusion, is deadlock-free, and avoids both atomic instructions and memory barriers (which avoid read-after-write anomalies) in all executions on TSO machines [3]. Although prwlock readers never contain explicit memory barriers, and thus might appear to violate this "law of order", prwlock uses IPIs to serialize reader execution with respect to writers, and IPI handling has the same effect as a memory barrier.

3.2 Basic Design

Consensus using bounded staleness: Prwlock introduces a 64-bit version variable (ver) to the lock structure. Each writer increases the version and waits until all readers see this version. As shown in Figure 2, ver creates a series of *happens-before* dependencies between readers and writers. A writer can only proceed after all readers have seen its new version. This ensures correct rwlock semantic on a machine with total-store order (TSO) consistency since a certain memory store can be visible only after all previous memory operations are visible.



```
                                      lock(writer);
while (writer != FREE) {
                                      ver++;
    status[my_id] = ver;
}                                     for_each (id) {
                                          while(status[id] < ver);
                                      }
```

Figure 2: Simple reader-writer lock with version report

However, there are still several issues with such an approach. First, a writer may never be able to enter the write-side critical section if a supposed reader never enters the read-side critical section again. Second, a reader may migrate from one core to another core so that the departing core may not be updated. Hence, such an approach may lead to arbitrarily lengthy latency or even starvation in the write side.

Handling straggling readers: To address the above issues, prwlock introduces a message-based consensus protocol to let the writer actively send consensus requests to readers when necessary. The design is motivated by the relatively small cost for message passing in contemporary processors. Hence, prwlock uses IPIs to request straggling readers to immediately report their status.

This design solves the straggling reader problem. However, if a reader is allowed to sleep in a read-side critical section, a sleeping reader may miss the consensus request so that a writer may be blocked infinitely.

Supporting sleeping readers: To address the sleeping reader issue, prwlock uses a hybrid design by combining the above mechanism with traditional counter-based rwlocks. Prwlock tracks two types of readers: passive and active ones. A reader starts as a passive one and does not synchronize with others, and thus requires no memory barriers. A passive reader will be converted into an active one before sleeping. A shared counter is increased during this conversion. The counter is later decreased after an active reader released its lock. Like traditional rwlocks, the writer uses this counter to decide if there is any active reader.

As sleeping in reader-side critical section is rare, prwlock enjoys good performance in the common case, yet still preserves correctness in a rare case where there are sleeping readers.

3.3 Prwlock Algorithms

Figure 3 and Figure 4 show a skeleton of the read-side and write-side algorithms of prwlock. For exposition simplicity, we assume that there is only one lock and preemption is disabled within these functions so that they can use per-cpu states safely.

Read-side algorithm: Passive readers are tracked distributively by a per-core reader status structure (**st**), which remembers the newest seen version and the passive status of a prwlock on each core. A reader should first set its status to *PASSIVE* before checking the writer lock, or there would be a time window at which the reader has already seen that the writer lock is free but has not yet acquired the reader lock. If the consensus messages (e.g., IPI) were delivered in this time window, the writer could also successfully acquire the lock and enter the critical section, which would violate the semantic of rwlock. If the reader found that this lock is writer locked, it should set its status back to *FREE*, wait until the writer unlocks and try again (line 4-8).

Depending on the expected writer duration, prwlock could either choose to spin on the writer status, or put the current thread to sleep. In the latter case, reader performance largely depends on the sleep/wakeup mechanism (section 4).

If a reader is holding a lock in *passive* mode while being scheduled out, the lock should be converted into an active one by increasing the active counter (*ScheduleOut*). To unlock a reader lock, one just needs to check whether the lock is held in passive mode and unlock it accordingly (*ReadUnlock*).

Hence, no atomic instructions/memory barriers are necessary in reader common paths on TSO architectures. Moreover, readers do not communicate with each other as long as they remain PASSIVE, thus guaranteeing perfect reader scalability and low reader latency.

Write-side algorithm: Writer lock acquisition can be divided into two phases. A writer first locks the writer mutex and increases the version to enter phase 1 (line 6-20). Then it checks all online cores in the current domain to see if the core has already seen the latest version. If so, it means that reader is aware of the writer's intention, and will not acquire reader lock until the writer releases the lock. For cores not seeing the newest version, the writer sends an IPI and asks for its status. Upon receiving an IPI, an unlocked reader will report to the writer by updating its local version (*Report*). A locked reader will report later after it leaves the read-side critical section or falls asleep. After all cores have reported, the consensus is done among all passive readers. The writer then enters phase 2 (line 21-23). In this phase, the writer simply waits until all active readers exit. For a writer-preference lock, a writer can directly pass the lock to a pending writer, without achieving a consensus again (line 1-2 in *WriteUnlock* and line 2-4 in *WriteLock*).

```
Function ReadLock(lock)
1 st ← PerCorePtr(lock.rstatus, CoreID);
2 st.reader ← PASSIVE;
3 while lock.writer ≠ FREE do
4     st.reader ← FREE;
5     st.version ← lock.version;
6     WaitUntil(lock.writer == FREE);
7     st ← PerCorePtr(lock.rstatus, CoreID);
8     st.reader ← PASSIVE;
9 /* Barrier needed here on non-TSO architecture */;
```

```
Function ReadUnlock(lock)
1 st ← PerCorePtr(lock.rstatus, CoreID);
2 if st.reader = PASSIVE then
3     st.reader ← FREE;
4 else
5     AtomicDec(lock.active);
6 /* Barrier needed here on non-TSO architecture */;
7 st.version ← lock.version;
```

```
Function ScheduleOut(lock)
1 st ← PerCorePtr(lock.rstatus, CoreID);
2 if st.reader = PASSIVE then
3     AtomicInc(lock.active);
4     st.reader ← FREE;
5 st.version ← lock.version;
```

Figure 3: Pseudocode of reader algorithms

Example: The right part of Figure 5 shows the state machine for prwlock in the reader side. A reader in passive mode may switch to the active mode if the reader goes to sleep. It cannot be directly switched back to passive mode until the reader releases the lock. The following acquisition of the lock will be in passive mode again.

The left part of Figure 5 shows an example execution

Function WriteLock(lock)

```
lastState ← Lock(lock.writer);
if lastState = PASS then
    return;
    /* Lock passed from another writer */
newVersion ← AtomicInc(lock.version);
coresWait ← ∅;
for ID ∈ AllCores do
    if Online(lock.domain, ID) ∧ ID ≠ CoreID then
        if PerCorePtr(lock.rstatus, CoreID).version ≠
        newVersion then
            AskForReport(ID);
            Add(ID, coresWait);
for ID ∈ coresWait do
    while PerCorePtr(lock.rstatus, CoreID).version ≠
    newVersion do
        Relax();
while lock.active ≠ 0 do
    Schedule();
```

Function WriteUnlock(lock)

```
if SomeoneWaiting(lock.writer) then
    Unlock(lock.writer, PASS);
else
    Unlock(lock.writer, FREE);
```

Function Report(lock)

```
st ← PerCorePtr(lock.rstatus, CoreID);
if st.reader ≠ PASSIVE then
    st.version ← lock.version;
```

Figure 4: Pseudocode of writer algorithms

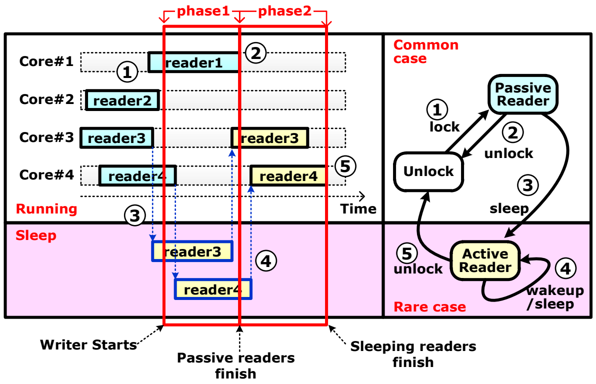


Figure 5: An example execution of readers (left) and the state machine of reader (right). Writer is not shown here.

of readers and how the consensus is done. Before a writer starts to acquire the lock, reader2 has finished its read critical section, while reader3 sleeps in its read critical section due to waiting for a certain event. Reader1 and reader4 have just started their read critical sections but have not finished yet.

In phase 1, there is a writer trying to acquire the lock in write mode, which will increase the lock version and block all upcoming readers. It will send IPIs to current active readers that have not seen the newest lock version. If reader2 in core2 has done a context switch and another thread is running right now, no IPI is required for core2. Reader4 in core4 may go to sleep to wait for a certain event, which will switch to be an active reader. No IPI is required for core4 as there is no reader in core4 at that time. At the end of phase1, all passive readers have left the critical sections. Thus, in phase 2, the writer waits all active readers to finish their execution and finally the lock can be granted in write mode. For a writer-preference prwlock, the writer can directly pass the lock to next writer, which can avoid unnecessary consensus among readers for consecutive writers.

Correctness on TSO architecture: The main difference between rwlocks and other weaker synchronization primitives is that rwlocks enface a strong visibility guarantee between readers and writers. This is guaranteed in prwlock with the help of TSO consistency model.

Once a reader sees an FREE prwlock, we can be sure that: 1) That FREE was set by the immediate previous writer, as writers will always ensure all reader see its LOCKED status before continuing; 2) As memory writes become visible in order under TSO architectures, updates made by the previous writer should also be visible to that reader. The same thing goes with earlier writers; 3) A writer must wait until all readers to see it, so no further writers can enter critical section before this reader exits. Thus prwlock ensures a consistent view of shared states.

These three properties together guarantee that a reader should always see the newest consistent version of shared data protected by prwlock. Moreover, as all readers explicitly report the newest version during writer lock acquisition, writers are also guaranteed to see all the updates (if any) made by readers to other data structures.

On non-TSO architectures, two additional memory barriers are required in reader algorithm as marked in Figure 3. The first one ensures that readers can see the newest version of shared data after acquiring the lock in the fast path. The second one makes readers' memory updates visible to the writer before releasing reader locks.

3.4 OS Kernel Incorporation

There are several issues in incorporating prwlock to an OS kernel. First, the scope of a prwlock could be either global or process-wide and there may be multiple prwlocks in each scope. Each prwlock could be shared by multiple tasks. To reduce messages between readers and writers, prwlock uses the *lock domain* abstraction to group a set of related prwlocks that can do consensus together. A domain tracks CPU cores that are currently executing tasks related to a prwlock. Currently, a domain could be process-wide or global. We now describe how prwlock uses the domain abstraction:

Domain Online/Offline: It is possible that the scope for a set of prwlocks may be switched off during OS execution. For example, for a set of locks protecting the address space structure for a process, the structure may be switched off during an address space switch. In such cases, prwlock uses the domain abstraction to avoid unnecessary consensus messages. A domain maintains a mapping from cores to its online/offline status. Only CPU cores within an active domain will necessitate the sending of messages. Figure 6 shows how to dynamically adjust the domain. The algorithm is simple as the consensus protocol can tolerant inaccurate domains.

When a domain is about to be online on a core, it simply sets the mapping and then performs a memory barrier (e.g., mfence). As the writer always sets its status before checking domains, it is guaranteed that either a writer could see the newly online core, or incoming readers on that core can see the writer is acquiring a lock. In either case, the rwlock semantic is maintained. To correctly make a domain offline from a core, a memory barrier is also needed before changing the domain to ensure that all previous operations are visible to other cores before offline.

Currently, for domains that correspond to processes, prwlock makes domains online/offline before and after context switches. However, it is possible to make a domain offline at any time if readers are expected to be infrequent afterward. When outside a domain, readers must acquire all prwlocks in the slower *ACTIVE* state. We choose to leave the choice to lock users as they may have more insight on the workload.

```
Function DomainOnline(dom)
coreSt ← PerCorePtr(dom.cores, CoreID);
coreSt.online = TRUE;
MemoryBarrier();
```

```
Function DomainOffline(dom)
coreSt ← PerCorePtr(dom.cores, CoreID);
MemoryBarrier();
coreSt.online = FALSE;
```

Figure 6: Domain management algorithms

Task Online/Offline: A task (e.g., a thread) may be context switched to other tasks and a task may also be migrated from one core to another core. prwlock uses task online/offline to handle such operations. When a task is about to be switched out while holding a prwlock in PASSIVE mode, it will change its lock status to be ACTIVE and increase the active reader counter if it previously holds a prwlock in passive read mode. This makes sure that a writer will wait until this task is scheduled again to leave its critical section to proceed. A task needs to do nothing when it is scheduled to be online again.

Downgrade/Upgrade: Typical operating systems usually support downgrading an rwlock from write mode to read mode and upgrading from read mode to write mode. Prwlock similarly supports lock downgrading by setting the current task to be in read mode and then releasing the lock in write mode. Unlike traditional rwlocks, upgrading a prwlock from read mode to write mode may be more costly in a rare case when the upgrading reader is the only reader, due to the lack of exact information regarding the number of readers. To upgrade a lock from read to write mode, prwlock tries to acquire the lock in write mode in the read-side critical section, but counts one less readers (excluding the upgrading reader itself) when acquiring the lock.

3.5 User-level Support

While it is straightforward to integrate prwlock in the kernel, there are several challenges to implementing it in user space. The major obstacle is that we cannot disable preemption during lock acquisition at user space. That is to say, we can no longer use any per-core data structure, which makes the algorithm in Figure 3 impossible.

To solve this problem, prwlock instead relies on some kernel support. The idea behind is simple: when it is necessary to perform any operation on per-core state, prwlock enters kernel and lets kernel handle it.

Instead of using a per-core data structure to maintain passive reader status, we introduce a per-thread data structure in user space. Each thread should register an instance of it to the kernel before performing lock operations, since there is only one thread running on each core at any time. Such per-thread data structures resemble a per-core data structure used in the kernel algorithm.

For performance considerations, the reader critical paths should be entirely in user space, or the syscall overhead would ruin prwlock's advantage of short latency. As a user application may be preempted at any time, our reader lock may experience several TOCTTOU problems. Recall that in prwlock a passive lock is maintained in per-core status while active locks are maintained in the shared counter; checking and changing the passive lock mode should be done atomically.

For example, line 2-3 of ReadUnlock algorithm in Figure 7 check if a reader is a passive one, and if so, release the passive lock by setting status to FREE. If the thread is preempted between line 2 and line 3, the lock might be converted into an active lock and the active count is increased. When it is later scheduled, the active count will not be decreased since the decision has already been made before. As a result, the rwlock becomes imbalanced and a writer can never acquire the lock again.

To overcome this problem, we add a preemption detection field into the per-thread data structure. As is shown in Figure 7, the reader first sets the status to PASSIVE and checks if it has been preempted while locking passively. If so, it decreases the active counter since the lock is now an active lock.

Function ReadUnlock(lock) for user-level prwlock

```
st ← PerThreadPtr(lock.rstatus);
st.reader ← FREE;
if st.preempted then
    AtomicDec(lock.active);
    st.preempted ← FALSE;
st.version ← lock.version;
```

Function ScheduleOut(lock)

```
st ← PerThreadPtr(lock.rstatus);
if st.reader = PASSIVE then
    AtomicInc(lock.active);
    st.preempted ← TRUE;
    st.reader ← FREE;
st.version ← lock.version;
```

Figure 7: Pseudocode of unlock algorithm with preemption detection

For the write-side algorithm, since it is not possible to send IPIs in user space, almost all writers should enter kernel to acquire the lock. Fortunately, mode switch cost between kernel and user space (around 300 cycles) is typically negligible compared to writer lock acquisition time (usually more than 10,000 cycles).

3.6 Performance Analysis

Memory barrier: In the common path of read-side critical section, prwlock requires no memory barrier when there is no outstanding writer. The only memory barrier required is when a CPU core is about to leave a lock domain, e.g., switch to another task and make current lock domain offline or online. However, domain online/offline operations are rare in typical execution. Hence, prwlock enjoys good performance scalability in common cases.

Writer cost: It appears that using IPIs may significantly increase the cost of writes, due to the IPI cost, possible mode switches and disturbed reader execution. However, the cost of IPIs and mode switches are small and usually in the scale of several hundreds to one thousand cycles. Further, as a writer usually needs to wait for a while until all readers have left the critical section, such costs can be mostly hidden. Though there may be a few cold cache misses due to disturbing reader execution, such misses on uncontended cache lines would be much smaller than the contention on shared states between readers and writers in traditional rwlocks.

In contrast to traditional rwlocks, the more readers are currently executing in the read-side critical section, the faster that a write can finish the consensus and get the lock in write mode (section 6.2.4). This is because readers will likely see the writer, and thus report immediately. Such a feature fits well with the common usage of rwlocks (more readers than writers).

Space overhead: Since prwlock is essentially a distributed rwlock, it needs $O(n)$ space for a lock instance. More specifically, current implementation needs 12 bytes (8 for version and 4 for reader status) per core per lock in order to maximize performance. It is also possible to pack a 7 bit version and a 1 bit status into one byte to save space. Another several bytes are needed to store writer status, whose exact size depends on the specific writer synchronization mechanism used. Further, an additional 1 byte per core is needed to store domain online status to support the lock domain abstraction.

By using the Linux kernel's per-cpu storage mechanism, a lock's per-cpu status could be packed into the same cache line as other per-cpu status words. Compared with other scalable rwlock algorithms (e.g. brlock, SNZI rwlock, read-mostly lock), prwlock imposes similar or lower space overhead.

Memory consistency model requirement: As prwlock relies on a series of happened-before relationship of memory operations, it requires that memory store operations are executed and become visible to others in issuing order (TSO consistency). Fortunately, this assumption holds for many commodity processor architectures like x86(64), SPARC and zSeries.

4 Decentralized Parallel Wakeup

Issues with centralized sequential wakeup: Sleep/wakeup is a common OS mechanism that allows a task to temporarily sleep to wait until a certain event happens (e.g., an I/O event). Operating systems such as Linux, FreeBSD and Solaris use a shared queue to hold all waiting tasks. It is usually the responsibility of the signaling task to wake up all waiting tasks. To do this, the signaling task first dequeues the task from the shared task queue, and then does something to prepare waking up the task. Next, the scheduler chooses a core for the task and inserts the task to the percpu runqueue. Finally, the scheduler sends a rescheduling IPI to the target core so that the awakened task may get a chance to be scheduled. The kernel will repeat sending IPIs until all awakened tasks have been rescheduled.

There are several issues with such a centralized, sequential wakeup mechanism. First, the shared waiting queue may become a bottleneck as multiple cores trying to sleep may contend on the queue. Hence, our first step involves using a lock-free wakeup queue so that the lock contention can be mitigated. However, this only marginally improves performance.

Our further investigation uncovers that the main performance scalability issue comes from the cascading wakeup phenomenon, as shown in Figure 8. When a writer leaves its write-side critical section, it needs to wake up all readers waiting for it. As there are multiple readers sleeping for the writer, the writer wakes up all readers sequentially. Hence, the waiting time grows

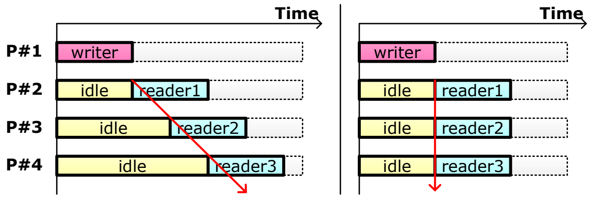


Figure 8: Issue with centralized, sequential wakeup (left) and how decentralized parallel wakeup solve this problem (right).

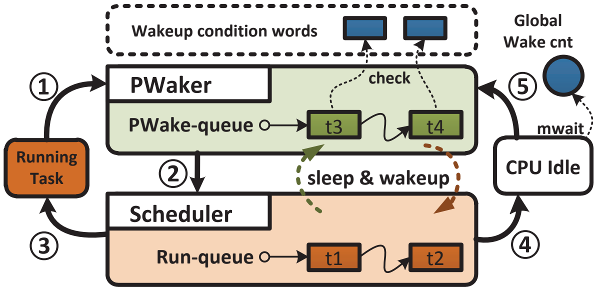


Figure 9: Key data structure and state transition graph of decentralized parallel wakeup in each core.

linearly with the number of readers.

Decentralized parallel wakeup: To speed up this process, prwlock distributes the duty of waking up tasks among cores. In general, this would risk priority inversion, but all prwlock readers always have equal priority.

Figure 9 shows the key data structure used in the decentralized parallel wakeup. Each core maintains a per-core wakeup queue (PWake-queue) to hold tasks sleeping on such a queue, each of which sleeps on a wakeup condition word. When a running task is about to sleep (step 1), it will be removed from the per-cpu runqueue and inserted to the per-cpu wakeup queue. Before entering the scheduler, if the kernel indicates that there is a pending request (e.g., by checking the wakeup counter), each core will first peek the PWake-queue to see if there is any task to wake up by checking the status word. If so, it will then insert the task to runqueue. This may add some cost to the per-cpu scheduler when there are some pending wakeup requests. However, as there are usually only very few tasks waiting in a single core, the cost should be negligible. Further, as all operations are done locally in each core, no atomic instructions and memory barriers are required. Finally, as a task generally wakes up on the core that last executed it, this task may benefit from better locality in both cache and TLBs. After checking the PWake-queue, each core will execute its scheduler (step 2) to select a task to execute (step 3).

As the new wakeup mechanism may require a core to poll the wakeup queue to reschedule wakeup tasks in the per-core scheduler, it may cause waste of power when there are no runnable tasks in a processor. To address this problem, our wakeup mechanism lets each idle core use the mwait mechanism[1] to sleep on a global word (step 4). When a writer finishes its work and signals to wake up its waiting tasks, the writer touches the word to wake up idle cores, which will then start to check if any tasks in the wakeup queue should be wakened up.

5 Implementation and Applications

We have implemented prwlock on several versions of Linux, and integrated it with the Linux virtual memory system by replacing the default rwlock. The porting effort among different versions of Linux is trivial and one student can usually finish it in less than one hour.

Linux address space: As prwlock is still an rwlock, it can trivially replace the original rwlock in Linux virtual memory subsystem. We write a script to replace more than 600 calls to *mmap_sem*. We add several hooks to process fork, exec, exit, wakeup and context switch. The prwlock library comprises of less than 300 LoC and requires manual change of less than 30 LoC other than the automatically replaced calls to *mmap_sem*. This is significantly less than the prior effort (around 2,600 LoC for page fault handling on anonymous memory mapping only) [10], yet with a complete replacement.

User-level prwlock and RCU: We have also implemented user-level prwlock, which comprise about 500 LoC. We further used the consensus protocol of prwlock to implement quiescence detection to implement a user-level RCU; this has better read-side throughput and faster quiescence detection than previous user-level quiescence detection mechanisms (section 6.3). We modified a famous database system named Kyoto Cabinet [17], by replacing a rwlock with prwlock to protect its data tables.

6 Evaluation

6.1 Evaluation Setup

Kernel prwlock: We use three workloads that place intensive uses of virtual memory: Histogram [24], which is a MapReduce application that counts colors from a 16GB bitmap file; Metis [19] from MOSBENCH [6], which computes a reverse index for a word from a 2GB Text file residing in memory; and Psearchy [6], a parallel version of searchy that does text indexing. They represent different intensive usages of the VM system, whose ratio between write (memory mapping) and read (page fault) are small, medium and large. We also implemented a concurrent hashtable [25] in kernel as a micro-benchmark to characterize prwlock and its alternatives.

User-space prwlock: We use several micro-benchmarks to compare prwlock with several alternatives like brlock and user-level RCU. As prwlock has a user-level RCU library, we also compare its performance

[1] mwait/monitor are x86 instructions that setup and monitor if an memory location has been touched by other cores.

to traditional signal-based user space RCU [14]. To show that prwlock can scale up user-space applications, we also evaluated the Kyoto Cabinet database using prwlock and the original rwlock.

As the performance characteristic that prwlock relies on are similar for Intel and AMD machines, we mainly run our tests on a 64-core AMD machine, which has four 2.4 GHZ 16-core chips and 128 GB memory. For each benchmark, we evaluate the throughput in a fixed time and collect the arithmetic mean of five runs.

6.2 Kernel-level prwlock

6.2.1 Application Benchmarks

We compare the performance of prwlock with several alternatives, including the default rwlock in Linux for virtual memory, percpu read-write lock [5], and an RCU-based VM design [10] (RCUVM). We are not able to directly compare prwlock with brlock as it has no sleeping support. As RCUVM is implemented in Linux 2.6.37, we also ported prwlock to Linux 2.6.37. As different kernel versions have disparate mmap and page fault latency, we use the Linux 2.6.37 kernel as the baseline for comparison. For the three benchmarks, we present the performance scalability for Linux-3.8 (L38), percpu-rwlock (pcpu-38) and prwlock on Linux 3.8 (prw-38), as well Linux 2.6.37 (L237), RCUVM (rcu) and prwlock on Linux 2.6.37 (prw-237) accordingly.

Histogram: As histogram is a page-fault intensive workload and the computation is very simple, it eventually hits the memory wall after 36 cores on Linux 3.8 for both percpu-rwlock and prwlock, as shown in Figure 10. Afterwards, both prwlock and percpu-rwlock show similar performance thrashing, probably due to memory bus contention. Percpu-rwlock scales similarly well and is with only a small performance gap with prwlock; this is because both have very good read-side performance. In contrast, the original Linux cannot scale beyond 12 cores due to contention on *mmap_sem*. As a result, prwlock outperforms Linux and percpu-rwlock by 2.85X and 9% respectively on 64 cores.

It was quite surprising that prwlock significantly outperforms RCUVM. This is because currently RCUVM only applies RCU to page fault on anonymous pages, while histogram mainly faults on a memory-mapped files. In such cases, RCUVM retries page fault with the original *mmap_sem* and thus experiences poor performance scalability. Though RCUVM can address this problem by adding RCU support for memory-mapped files, prwlock provides a much easier way to implement and reason about correctness due to its clear semantic.

Metis: Metis has relatively more mmap operations (mainly to allocate memory to store intermediate data), but is still mainly bounded by page fault handling on anonymous memory mapping. As shown in Figure 11, prwlock performs near linearly to 64 cores with a speedup over percpu-rwlock and original Linux by 27% and 55% in 64 cores accordingly. This is mainly due to scalable read-side performance and small write-side latency. There is a little bit performance gap with RCUVM, as RCUVM further allows a writer to proceed in parallel with readers.

Psearchy: Psearchy has many parallel mmap operations from multiple user-level threads, which not only taxes page fault handler, but also mmap operations. Due to extended mmap latency, percpu-rwlock cannot scale beyond 4 cores, as shown in Figure 12. In contrast, prwlock performs similarly with Linux before 32 cores and eventually outperforms Linux after 48 cores, with a speedup of 20% and 5.63X over Linux and percpu-rwlock for Linux 3.8. There is a performance churn between 32 and 48 cores for Linux, probably due to the contention pattern changes during this region. For Linux 2.6.37 with smaller mmap latency, prwlock performs similarly with Linux under 48 cores and begins to outperform Linux afterwards. This is due to the contention over rwlock in Linux, while prwlock's excellent read-side scalability makes it still scale up.

As psearchy is a relatively mmap-intensive workload, prwlock performs worse than RCUVM as RCUVM allows readers to proceed in parallel with writers. Under 64 cores, prwlock is around 6% slower than RCUVM. Psearchy can be view as a worst case for prwlock and we believe this small performance gap is worthwhile for much less development effort.

6.2.2 Benefits of Parallel Wakeup

Figure 13 using the histogram benchmark to show how parallel wakeup can improve the performance of both RCUVM and original Linux. Parallel wakeup boosts RCUVM by 34.7% when there are multiple readers waiting. prwlock improves the performance of original Linux by 47.6%. This shows that parallel wakeup can also be separately applied to Linux to improve performance.

We also collected the mmap and munmap cost for both Linux and prwlock, which are 934us, 1014us and 567us, 344us. With the fast wakeup mechanism, the cost for Linux has decreased to 697us and 354us.

6.2.3 Benefits of Eliminating Memory Barriers

We use a concurrent hashtable [25] to compare prwlock with RCU, rwsem and brlock. Figure 16 illustrates the performance. RCU has a nearly zero reader overhead and outperform all rwlocks. The throughput of rwsem vanishes because of cache contention. Thanks to elimination of memory barriers, prwlock shows higher throughput than brlock. More tests reveal that the lookup overhead mainly comes from cache capacity misses while accessing hash buckets. Prwlock's speedup over brlocks would

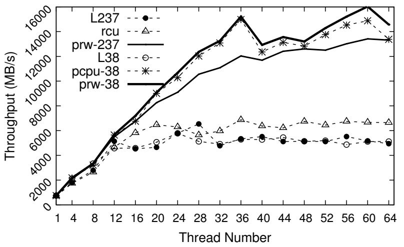


Figure 10: Histogram throughput scalability for original Linux, percpu-rwlock, prwlock on Linux 3.8

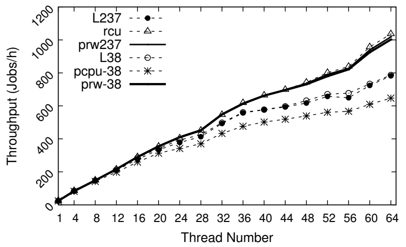


Figure 11: Metis throughput scalability for original Linux, percpu-rwlock, prwlock on Linux 3.8

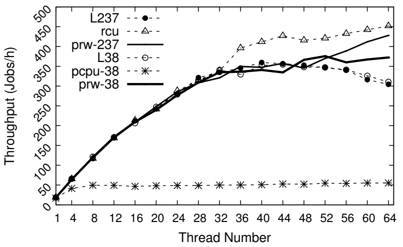


Figure 12: Psearchy throughput scalability for original Linux, percpu-rwlock, prwlock on Linux 3.8

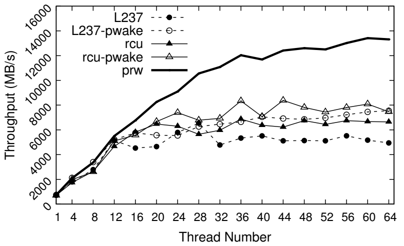


Figure 13: Benefit of parallel wakeup for Histogram.

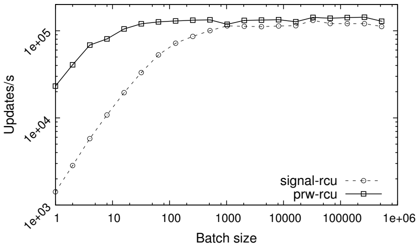


Figure 14: Update performance with batch size

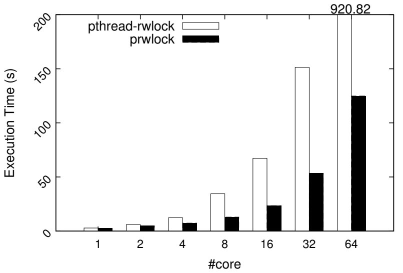


Figure 15: Benefit of prwlock for an in-memory DB

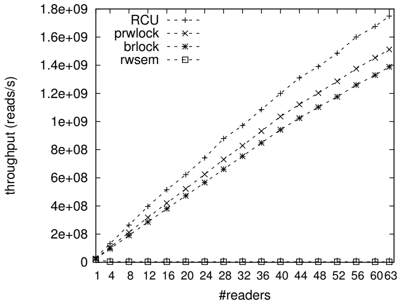


Figure 16: Lookup performance of hashtable

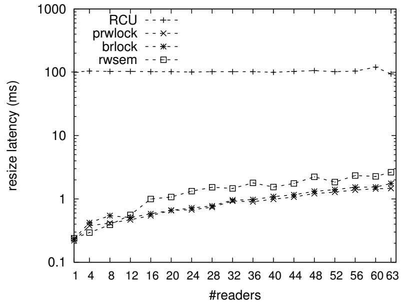


Figure 17: Resize latency of hashtable

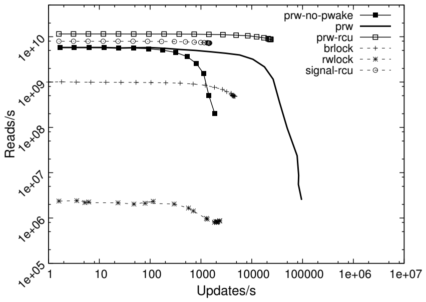


Figure 18: Relation between reader/writer throughput

be much larger if there was a cache hit (not shown here).

By using rwlocks instead of RCU, resizing the hashtable is much simpler and faster as all readers are blocked during resizing. Figure 17 presents the total latency to shrink and grow the hash table on different concurrency levels. Rwlocks shows up to two orders of magnitude shorter resizing latency compared to RCU. As hashtable resizes have negative impact on lookup performance, shorter resize latency is desirable to maintain a stable lookup performance. Prwlock only shows marginally better performance compared to other two rwlocks, as in this test most of the time is spent in critical section rather than writer lock acquisition.

6.2.4 *Critical section efficiency*

To better characterize different rwlocks, we also evaluate their raw critical section overhead (lock/unlock pair latency), which is shown in Table 3. prwlock shows best reader performance as its common path is simple and has no memory barriers. It is interesting that prwlock has much higher writer latency when there is no reader, since the writer has to use IPIs to ask every online core to report. Though rmlock (Read-Mostly Lock in FreeBSD) also eliminates memory barriers in reader common paths, its reader algorithm is more complex than prwlock, and thus results in higher reader latency. Writer of rwsem (Linux's rwlock) performs well for few readers, but suffers from contention with excessive readers.

	brlock	rmlock	rwsem	prwlock
Reader latency (1 reader)	58	46	107	12
Reader latency (64 readers)	58	46	20730	12
Writer latency (0 reader)	17709	136	100	65511
Writer latency (63 readers)	89403	622341	3235736	6322

Table 3: Critical section efficiency (average of 10 millions runs)

6.3 User-level Prwlock

Figure 18 shows the impact of writer frequency on reader throughput for several locking primitives, by running 63 reader threads and 1 writer thread. Writer frequency is controlled by varying the delay between two writes, which is similar done as Desnoyers et al. [14]. Note that 1 writer is the worst case of prwlock since if there is more than 1 writer, the writer lock could be passed among writers without redoing consensus. To compare the time for a consensus, we fixed the batch size of both RCU algorithms to 1. That means they must wait a grace period for every update.

Prwlock achieves the highest writer rate. This confirms that our version-based consensus protocol is more efficient than prior approaches. Prwlock's read side performance is similar to RCU, and notably outperforms brlock, mainly because prwlock requires no memory barriers in reader side. Parallel wakeup also contributes to prwlock's superior performance. Since it improves reader concurrency, prwlock is able to achieve higher reader throughput when there are many writers. Writer performance is also greatly improved since wakeup is offloaded to each core.

We can also notice that prwlock-based RCU performs consistently better than the signal-based user-level RCU. Thanks to prwlock's kernel support, the reader-side algorithm of prwlock RCU is simpler, which results in a higher reader throughput. Besides, prwlock-RCU has orders of magnitude higher writer rate than signal-based RCU, due to its fast consensus protocol.

We further vary the batch size to study RCU performance, as shown in Figure 14. Prwlock-RCU reaches its peak performance before the batch size reaches 100 and performs much better when the batch size is less than 1000. Small batch size helps control the memory footprint since it allows faster reclamation of unused objects.

Kyoto Cabinet: Figure 15 shows the improvement of prwlock over using the original pthread-rwlock. As the workload for different number of cores is different, the increasing execution time with core does not mean poor scalability. For all cases, prwlock outperforms original rwlock and the improvement increases with core count. Under 64 cores, prwlock outperforms pthread-rwlock by 7.37X (124.8s vs. 920.8s). The reason is that the workload has hundreds of millions read accesses and pthread-rwlock incurs high contention on the shared counter, while prwlock places no contention in the reader-side.

7 Conclusions and Future Work

This paper has described passive reader-writer lock, a reader-writer lock that provides scalable performance for read-mostly synchronization. Prwlock can be implemented in both kernel and user mode. Measurements on a 64-core machine confirmed its performance and scalability using a set of application benchmarks that contend kernel components as well as a database. In future work, we will investigate and optimize prwlock in a virtualized environment (which may have higher IPI cost).

Acknowledgment

We thank our shepherd Eddie Kohler and the anonymous reviewers for their insightful comments and suggestions. This work is supported in part by China National Natural Science Foundation (61003002), Shanghai Science and Technology Development Funds (No. 12QA1401700), a foundation for the Author of National Excellent Doctoral Dissertation of China, and Singapore CREATE E2S2.

References

[1] Linux test project. http://ltp.sourceforge.net/.

[2] Version-based brlock. https://www.kernel.org/pub/linux/kernel/people/marcelo/linux-2.4/include/linux/brlock.h.

[3] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *PPoPP*, 2011.

[4] A. Baumann, P. Barham, P. 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 *SOSP*, pages 29–44, 2009.

[5] S. S. Bhat. https://patchwork.kernel.org/patch/2157401/, 2013.

[6] S. Boyd-Wickizer, A. Clements, Y. Mao, A. Pesterev, M. Kaashoek, R. Morris, and N. Zeldovich. An analysis of Linux scalability to many cores. In *OSDI*, 2010.

[7] S. Boyd-Wickizer, M. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2012.

[8] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware reader-writer locks. In *PPoPP*, 2013.

[9] B. Cantrill and J. Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, 2008.

[10] A. Clements, M. Kaashoek, and N. Zeldovich. Scalable address spaces using RCU balanced trees. In *ASPLOS*, 2012.

[11] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys*, 2013.

[12] J. Corbet. Big reader locks. http://lwn.net/Articles/378911/.

[13] P. Courtois, F. Heymans, and D. Parnas. Concurrent control with readers and writers. *Comm. ACM*, 14(10), 1971.

[14] M. Desnoyers, P. McKenney, A. Stern, M. Dagenais, and J. Walpole. User-level implementations of read-copy update. *TPDS*, 23(2):375–382, 2012.

[15] D. Dice, V. J. Marathe, and N. Shavit. Lock cohorting: a general technique for designing NUMA locks. In *PPoPP*, 2012.

[16] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC*, pages 13–22, 2007.

[17] FAL Labs. Kyoto Cabinet. http://fallabs.com/kyotocabinet/.

[18] Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *SPAA*, 2009.

[19] Y. Mao, R. Morris, and M. F. Kaashoek. Optimizing MapReduce for multicore architectures. In *MIT Tech. Rep*, 2010.

[20] P. Mckenney. RCU usage in the Linux kernel: One decade later. http://www2.rdrop.com/users/paulmck/techreports/survey.2012.09.17a.pdf.

[21] P. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *Ottawa Linux Symposium*, 2001.

[22] J. Mellor-Crummey and M. Scott. Synchronization without contention. In *ASPLOS*, pages 269–278. ACM, 1991.

[23] D. Petrović, O. Shahmirzadi, T. Ropars, A. Schiper, et al. Asynchronous broadcast on the Intel SCC using interrupts. In *Many-core Applications Research Community Symposium*, 2012.

[24] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, pages 13–24, 2007.

[25] J. Triplett, P. E. McKenney, and J. Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *USENIX ATC*, 2011.

Large Pages May Be Harmful on NUMA Systems

Fabien Gaud
Simon Fraser University

Baptiste Lepers
CNRS

Jeremie Decouchant
Grenoble University

Justin Funston
Simon Fraser University

Alexandra Fedorova
Simon Fraser University

Vivien Quéma
Grenoble INP

Abstract

Application virtual address space is divided into pages, each requiring a virtual-to-physical translation in the page table and the TLB. Large working sets, common among modern applications, necessitate a lot of translations, which increases memory consumption and leads to high TLB and page fault rates. To address this problem, recent hardware introduced support for *large pages*. Large pages require fewer translations to cover the same address space, so the associated problems diminish.

We discover, however, that on systems with non-uniform memory access times (NUMA) large pages may fail to deliver benefits or even cause performance degradation. On NUMA systems the memory is spread across several physical nodes; using large pages may contribute to the imbalance in the distribution of memory controller requests and reduced locality of accesses, both of which can drive up memory latencies.

Our analysis concluded that: (a) on NUMA systems with large pages it is more crucial than ever to use memory placement algorithms that balance the load across memory controllers and maintain locality; (b) there are cases when NUMA-aware memory placement is not sufficient for optimal performance, and the only resort is to split the offending large pages. To address these challenges, we extend an existing NUMA page placement algorithm with support for large pages. We demonstrate that it recovers the performance lost due to the use of large pages and makes their benefits accessible to applications.

1 Introduction

Applications with large memory working sets require many virtual-to-physical address translations in page tables and TLBs. This drives up physical RAM consumption, increases TLB miss rate, and hurts performance [1][2][10]. According to one report, a large Oracle DBMS installation with 500 concurrent connections consumed 7GB of RAM for page tables alone! [5]. To address this problem, most modern hardware and OS introduced support for large pages. On x86 systems large pages are typically 2MB (512 times larger than regularly-sized 4KB pages), and support for 1GB pages is on the way[1]. Using larger pages requires fewer translations to cover the address space and diminishes the pressure on the TLB and physical memory.

While large pages are so crucial for performance of large-memory systems, they, unfortunately, also have downsides. Previous work reported and addressed increased memory footprints and physical memory fragmentation [13]. In this work, we report on a new problem: ***large pages hurt performance on NUMA systems***.

Modern NUMA systems are comprised of several *processor nodes* each containing a multicore CPU and a local DRAM, all inside a single physical server. The nodes are connected by the high-speed interconnect into a cache-coherent system, forming an abstraction of a single globally addressable memory. While CPUs can transparently allocate and access the memory on any node, accesses to remote nodes traverse the interconnect and access a remote memory controller, incurring higher latency and contributing to congestion on the interconnect. To achieve good performance on NUMA systems, we need to (1) maximize the fraction of memory accesses going to local nodes and (2) balance the traffic across the nodes and interconnect links. Unbalanced distribution of memory requests can increase the memory access latency on the overloaded controller to as many as 1000 cycles, compared to about 200 cycles on a not overloaded controller [6].

In this paper we show that ***large pages can exacerbate harmful NUMA effects, such as poor locality and imbalance***. Using large pages makes the unit of mem-

[1] 1GB pages are already supported by the hardware; support by the OS is still nascent, so few applications are able to use them at the time of the writing.

ory management (a page) more coarse. As a result, it is more likely that many frequently accessed memory addresses happen to map to the same physical page and overload the memory node hosting it – the so-called *hot page effect*. The hot-page effect cannot be addressed by page migration and balancing; page splitting must be performed prior to any attempts to rebalance memory. Likewise, large pages lead to more frequent *page-level false sharing* among threads, where threads access *different* data on the *same* page. False sharing leads to poor locality, which cannot be addressed by page migration alone.

In this work we:

- Quantify the performance degradation due to large pages on NUMA systems. We find that they affect between 25% and 30% of applications in our benchmark set and cause degradations between 5% and 43%.
- Demonstrate that these performance losses are due to NUMA factors, such as poor locality and imbalance.
- Show that the problem can be addressed using a combination of old and new techniques.

Our solution consists of two components: an existing NUMA-aware page placement algorithm Carrefour [6], and large-page extensions to Carrefour: *Carrefour-LP*. For some of the affected applications Carrefour alone is able to recover the lost performance, but in other cases Carrefour is ineffective due to the hot-page effect and page-level false sharing.

Even though hot pages and false sharing touched only a couple of benchmarks in our set, these effects will become pervasive on systems with much larger pages (e.g., 1GB), which are becoming common. Therefore, we implemented Carrefour-LP which addresses these problems by dynamically splitting large pages as needed. For applications affected by hot pages and false sharing, Carrefour-LP improves performance by 10%-80% relative to Carrefour alone. Carrefour together with Carrefour-LP significantly diminish or completely eliminate the performance degradation introduced by large pages and improve performance of some applications by 2-3× relative to Linux with large pages.

The rest of the paper is structured as follows: Section 2 motivates the work by presenting performance effects of using large pages on NUMA systems, Section 3 presents the solution, Section 4 evaluates it, Section 5 discusses related work, and lastly Section 6 summarizes the paper.

2 Large Pages and Adverse NUMA Effects

2.1 Experimental platform

For our experiments, we used two different server-class machines. Machine A has two 1.7GHz AMD Opteron 6164 HE processors, with 12 cores per processor, and 64GB of RAM. The system is equally divided into four NUMA nodes (i.e., six cores and 12GB of RAM per node). Machine B has four AMD Opteron 6272 processors, each with 16 cores (64 cores in total), and 512GB of RAM. It has eight NUMA nodes – 8 cores and 64GB of RAM per node. Both machines have HyperTransport 3.0 interconnect links.

We are running on Linux 3.9 and are using Transparent Huge Pages (THP) for large page allocation[2]. THP works by backing allocations of anonymous memory with 2MB pages whenever possible. Other kinds of memory, such as memory mapped files, are unaffected by THP and use 4KB pages. THP also uses a kernel thread to periodically scan for free memory regions that are at least 2MB in size, which are then used to replace groups of existing 4KB pages.

We used several benchmark suites representing a variety of different workloads: the NAS Parallel Benchmarks suite which is comprised of numeric kernels, MapReduce benchmarks from Metis, SSCA v2.2 (a graph processing benchmark) with a problem size of 20, and SPECjbb. From the NAS benchmark suite we picked the benchmarks that ran for at least 15 seconds. The memory usage of the benchmarks ranges from 518MB for EP from the NAS suite to 34,291MB for IS from NAS.

2.2 Large Pages on Linux

Figure 1 compares the performance of 4KB pages and 2MB pages using THP. We can see that THP increases performance (by up to 109%) for several benchmarks on both machines (e.g. WC, WR, WRMEM, and SSCA), but also significantly decreases performance by as much as 43% in some cases. CG, UA, and SPECjbb are all negatively affected by THP. Therefore, 2MB pages are not universally beneficial and neither are 4KB pages, so there is no "one size fits all."

To understand this phenomenon, we recorded two metrics that represent the potential benefits of large pages: the number of L2 cache misses caused by page table walks (obtainable from hardware performance counters), and the maximum time spent in the page fault handler by any core. L2 misses due to page table walks is a good indicator for the effect of TLB misses on performance. We expect large pages to increase the TLB coverage and reduce page table sizes. As a result, we expect the number L2 cache misses due to page table walks to drop when we use large pages. Similarly, large pages will reduce the number of page faults for allocations and

[2] Linux also allows using large pages via *libhugetlbfs*, but the latter required recompiling applications and pre-allocating memory for large pages, which was inconvenient, and, moreover, did not perform better than THP in our experiments.

thus the time spent in the page fault handler.

We also monitored two metrics related to NUMA efficiency: the *local access ratio* (LAR), which is the percentage of accesses to local memory, and the *traffic imbalance* on the memory controllers. Traffic imbalance is defined as the standard deviation of the memory request rate across the controllers, expressed as the percent of the mean. For memory intensive applications, a low LAR and a high imbalance signify a NUMA issue.

Table 1 shows the profiling results for a subset of interesting applications. As expected, applications that benefited from 2MB pages in Figure 1 (WC and SSCA) have fewer L2 misses due to page table walks, and for WC significantly less time spent in the page fault handler. The effects can be dramatic. For example, with SSCA on machine A the percentage of L2 misses due to page table walks is decreased from 15% to 2% when using 2MB pages, which results in a 17% performance increase. WC, which experiences a similar decrease in L2 misses but also a large decrease in time spent on page faults, has its performance increased more than two-fold on machine B.

The two other profiled benchmarks – CG and UA – perform much worse with 2MB pages. The profiling reveals that the degradation is caused by NUMA effects. With CG and 4KB pages, the load on the memory controllers is almost perfectly balanced, but with 2MB pages the imbalance is 20% on machine A and 59% on machine B. For UA, the problem is that the LAR decreases when using large pages, from about 88% to around 66%.

SPECjbb presents an interesting case. While the data in Figure 1 suggests that it does not benefit from large pages, profiling reveals that using large pages actually decreases the percent of L2 misses due to page table walks. At the same time, SPECjbb suffers from NUMA issues: the imbalance rises from 16% to 39% with large pages. Therefore, SPECjbb *could benefit* from large pages if NUMA effects were reduced.

3 Solutions

The previous section demonstrated that using large pages may introduce NUMA issues, which may either degrade performance relative to small pages (as they did for CG and UA) or leave the performance unchanged but prevent an application from enjoying the benefits of large pages (as they did for SPECjbb). In this section we first demonstrate that using a NUMA-aware page placement algorithm eliminates the NUMA issues for some applications, motivating the use of NUMA-aware page placement with large pages.

We then identify two new problems that a placement algorithm unaware of large pages does not address: the hot-page effect and the page-level false sharing. These effects, while affecting only two applications in our experiments, will become especially important as much larger pages (e.g., 1GB) come into use. To address them, we introduce large-page extensions (LP) to an existing NUMA placement algorithm.

For clarity of presentation, from now on we will focus on those applications that experience NUMA issues when large pages are used. Specifically, if the LAR or the imbalance is made worse by more than 15% by using large pages as opposed to small ones on either machine, the application is selected for presentation, otherwise it is omitted. The selected applications are: CG.D, LU.B, UA.B, UA.C, matrixmultiply, wrmem, SSCA, SPECjbb. For completeness, and to demonstrate that our solutions do not hurt the applications they cannot help, we do include performance results for the excluded applications at the end of Section 4.

3.1 Page balancing is not enough

We used a NUMA-aware page balancing algorithm Carrefour, which was shown to perform better than other similar solutions [6]. Carrefour works by gathering access samples for memory pages and then choosing a host node for a page based on the samples. If all of the samples for a page originated from a single node, then the page is migrated to that node. If the samples came from multiple nodes, then the page is interleaved (i.e. migrated to a random node). Carrefour also includes thresholds based on hardware counters, so that it is only enabled if NUMA problems are detected such as when the local access ratio is low or the imbalance on memory controllers is high.

We ran Carrefour in the kernel configured with 2M pages (Carrefour-2M). Figure 2 shows the performance of Carrefour-2M compared to Linux with 2M pages (labeled as *THP*) relative to Linux with 4K pages (labeled as *Linux*). We observe that while Carrefour-2M does improve performance for some applications, it fails to solve the problem across the board. For SPECjbb, Carrefour-2M addresses the NUMA issue; as shown in Table 2 it restores the balance on memory controllers that was introduced by large pages and improves the LAR.

At the same time, Carrefour-2M fails to improve performance for UA and CG. To understand why, we show profiling data for these applications in Table 2. We report five metrics: the percentage of total accesses to the most used page (PAMUP), the number of hot pages (NHP) defined as pages comprising more than 6% of the total accesses[3], the percentage of memory accesses to pages

[3] In order to perfectly balance the load on a 8-node NUMA machine, each node must be the target of 12.5% of the total memory accesses. Thus, we consider that if a page represents more than half of this amount, it is likely to create imbalance.

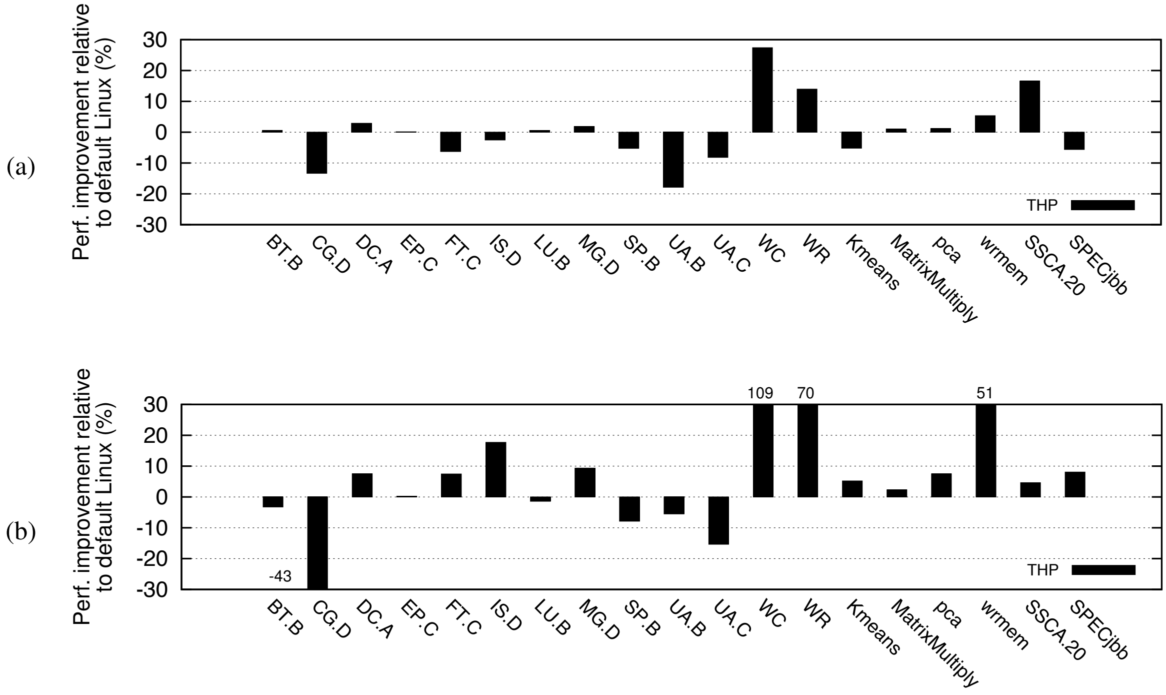


Figure 1: THP performance improvement over Linux on (a) machine A and (b) machine B. THP sometimes perform better than Linux, sometimes worse.

	Perf. incr. THP/4k (%)	Time spent in page fault handler (% of total time)		% L2 misses due to page table walk		Local access ratio (%)		Imbalance (%)	
		Linux	THP	Linux	THP	Linux	THP	Linux	THP
CG.D (B)	-43	2182ms (0.1%)	445ms (0%)	0	0	40	36	**1**	**59**
UA.C (B)	-15	102ms (0.2%)	53ms (0.1%)	0	0	**88**	**66**	14	12
WC (B)	109	**8731ms (37.6%)**	**3682ms (32.3%)**	**10**	**1**	50	55	147	136
SSCA.20 (A)	17	90ms (0%)	147ms (0.1%)	**15**	**2**	25	26	**8**	**52**
SPECjbb (A)	-6	8369ms (2.1%)	5905ms (1.5%)	**7**	**0**	12	15	**16**	**39**

Table 1: Detailed analysis of various application on machine A and B. The machine type is indicated in parentheses next to the name of the benchmark.

shared by at least two threads (PSP), the percentage of accesses to local memory (LAR), and the traffic imbalance on the memory controllers.

The results for CG reveal that there is a ***hot page problem***. Large pages cause the heavily accessed regions of the address space to be coalesced into a small number of hot pages (the PAMUP significantly increases), and because there are fewer hot pages than NUMA nodes it is impossible to balance them.

UA does not have a hot page issue, but it does have more pages that are shared among threads when large pages are used (the PSP significantly increases). This happens because each page holds more data and is thus more likely to contain data used by multiple threads. Since the threads do not share data, but share the *page*, we refer to this problem as ***page-level false sharing***. Carrefour-2M is then forced to interleave these pages whereas if there were less sharing the pages could be placed on the nodes where they are most heavily used for maximum locality. As a result, Carrefour-2M delivers a lower LAR than Linux with small pages.

In summary, Carrefour-2M is only able to address NUMA issues induced by large pages in cases where they are not caused by the hot-page effect and page-level false-sharing.

While these problems affected only two applications in our experiments, they will become pervasive as pages much larger than 2MB come into use. 1GB pages are already supported by the hardware; applications like large DBMS clearly motivate their use [5]. We did not evaluate 1GB pages, because they are poorly supported in Linux. 1GB pages are not compatible with THP, and while in

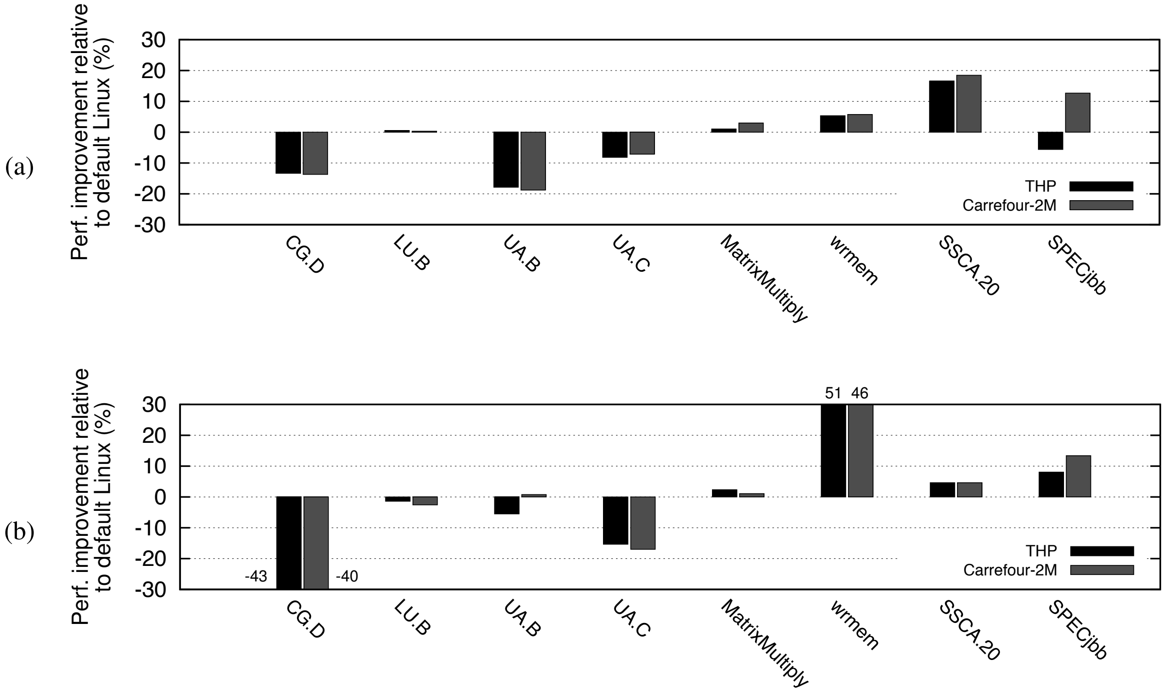


Figure 2: Performance improvement of Carrefour-2M and THP over Linux on applications whose NUMA metrics are affected by THP (2MB pages). Carrefour-2M is not always able to solve the problems for applications that suffer from THP.

theory it is possible to use them with lighugetlbfs, that has many challenges. First of all, the implementation is unreliable. We were not able to enforce the use of 1GB pages with NAS applications and observed many crashes with the Metis suite (because the latter uses a custom memory allocator). Second, the splitting of large pages, which is crucial to our solution, is not supported by libhugetlbfs and implementing it would require a significant effort.

However, since the use-case for very large pages is definitely there, they will become more common as the OS support improves. Then, the hot-page effect and page-level false sharing will become more common (Section 4.4 provides some preliminary data). To address these problems, we propose large-page extensions to Carrefour.

		Linux	**THP**	**Carrefour 2M**
SPECjbb	PAMUP	2%	6%	6%
	NHP	0	0	0
	PSP	**10%**	**36%**	**36%**
	Imbalance	**16%**	**39%**	**19%**
	LAR	26%	28%	27%
CG.D	PAMUP	**0%**	**8%**	**8%**
	NHP	**0**	**3**	**3**
	PSP	18%	34%	34%
	Imbalance	**0%**	**20%**	**20%**
	LAR	45%	45%	45%
UA.B	PAMUP	6%	6%	6%
	NHP	0	0	0
	PSP	**16%**	**70%**	**70%**
	Imbalance	9%	15%	17%
	LAR	**90%**	**61%**	**58%**

Table 2: Proportion of accesses to the most-used page (PAMUP) in %, number of hot pages (NHP), proportion of memory accesses to shared pages (PSP) in %, Imbalance in % and local access ratio (LAR) in % for Linux, THP and Carrefour-2M, on machine A (24 cores).

3.2 Carrefour-LP

Intuition suggests two basic solutions to the problem: *conservative* – prevent the problem by only creating large pages when necessary, or *reactive* – start with large pages and fix NUMA problems when they are observed. Each approach has potential benefits and drawbacks. The conservative approach can avoid NUMA related performance degradation but can also miss out on the benefits of large pages. On the other hand, the reactive approach will benefit from large pages, but must be able to quickly and accurately detect NUMA issues and must pay the overhead of fixing them.

We found that a good algorithm must be a combination of these approaches. The ***reactive component*** of our algorithm continuously monitors the hardware counters looking for the presence of NUMA effects under large pages, applies the page balancing techniques of Carrefour and splits the large pages if the latter are ineffective. The ***conservative component*** of the algorithm continuously monitors the virtual memory metrics and re-enables large pages if they are expected to deliver benefit but were previously disabled.

We also found that it is more practical and involves less overhead to enable large pages in the beginning and disable them later if they are deemed harmful. In particular, many applications have intensive memory-allocation phases at the very beginning of the program that suffer from lock contention if small pages are used.

Our full algorithm is presented in Algorithm 1. Lines 4-9 corresponds to the conservative component, the rest to the reactive component. The algorithm also details the hardware counter metrics that are being monitored. Since the monitoring is done continuously, the algorithm caters to phase changes in applications. Below we describe the rationale behind the decisions made in the algorithm.

3.2.1 Reactive component

The job of the reactive component is to disable large pages when they are harmful to the extent that even Carrefour-2M's page-balancing techniques cannot address the performance degradation. To that end, it estimates the local access ratio (LAR), a vital metric for detecting NUMA issues, with and without Carrefour and large pages.

We use AMD's instruction-based sampling (IBS)[4] to sample memory accesses to pages, and to learn whether the access was made from a local or a remote node. We only consider pages that have at least one sample where the access was serviced from DRAM, so that our decisions are not affected by pages that are easily cached. From the IBS samples, we estimate the LAR that would be obtained if the shared pages were migrated to a random node and if non-shared pages were migrated to the local node (i.e. interleaving and migrating pages with the Carrefour-2M algorithm). We also calculate the LAR that would be obtained if the same technique were used but with all of the 2MB pages split into 4KB pages.

Estimating the LAR for various what-if scenarios (e.g., if a page were migrated or if large pages were split into regular-sized) is trivial with IBS samples. IBS gives us data addresses and the node from which they were accessed. So we can compute the current LAR as well as the LAR that would be obtained if the pages where placed on different nodes. Similarly, we can map the data addresses to 4KB pages and compute the same metrics for the scenario if the large pages were split.

If, based on our estimates, the LAR can be improved by 15% with Carrefour-2M only and without splitting the pages, we simply run Carrefour-2M. Otherwise, if splitting pages would improve the LAR by at least 5%, then all shared 2MB pages are demoted into 4KB pages. Note that we are being cautious here: we try to address NUMA issues by page migration first, and split pages only if absolutely necessary. Splitting pages has overhead and may hurt applications that need them – hence our decision. In addition, large pages with more than 6% of the total accesses (hot pages, as defined in Section 3.1) are split and the constituent 4KB pages are interleaved.

This part of the algorithm relies on two thresholds. The first one is the 15% threshold used to decide whether we can improve the LAR simply by rearranging memory pages, without having to split large pages. That threshold was relatively easy to set across applications: the key is to use a relatively large number, since we want to be rather confident that we can improve performance without having to split pages. The second threshold, the 5% performance gain that we expect from splitting pages, needs to be any non-negligible number that would justify the splitting. Again, that threshold was relatively easy to tune across applications.

In the algorithm, we use the LAR computed per-application. Another option would be to use the LAR computed per-page, however this was difficult to do, because existing hardware monitoring facilities prevent us from obtaining enough samples to accurately compute per-page LAR (and even per-application LAR as explained in the next section). This is why the algorithm splits all 2MB pages when it detects the LAR can be improved.

3.2.2 Conservative component

The job of the conservative component is to re-enable large pages when they have been disabled but monitoring shows that they would be beneficial again. The conservative component uses two criteria to determine the benefit of large pages: the performance impact of TLB misses (based on the fraction of L2 misses caused by page table walks) and the maximum percentage of time any core spends processing page faults. The reason why we consider the time spent processing page faults is that large pages improve performance by decreasing this time. Indeed, soft page faults not only take CPU time, but also

[4] Intel systems have a similar facility called PEBS (Precise Event-Based Sampling).

Algorithm 1 Large-page Extensions to Carrefour

```
Enable 2MB page allocation and promotion
while true do
    Gather hardware performance counters and IBS samples for 1 sec
    if L2 misses due to page table walks $> 5\%$ then
        Enable 2MB page allocation
        Enable 2MB page promotion
    else if Max time spent on page faults $> 5\%$ then
        Enable 2MB page allocation
    end if
    if Estimated LAR improvement with only Carrefour $> 15\%$ then
        SPLIT_PAGES = false
    else if Estimated LAR improvement with Carrefour and splitting pages $> 5\%$ then
        SPLIT_PAGES = true
    end if
    if SPLIT_PAGES = true or 2MB page allocation is disabled then
        Split all shared 2MB pages into 4KB pages
        Disable 2MB page allocation
    end if
    Split and interleave 2MB hot pages
    Interleave and migrate pages with Carrefour
end while
```

incur costly synchronization [3]. The latter is the reason why we use the *maximum* fraction as opposed to the average: lock contention will be determined by the slowest core that holds page table locks.

The conservative component works as follows. If the impact of TLB misses is estimated to be greater than a threshold of 5%, then 2MB page allocation and 2MB page *promotion*[5] are both enabled via THP. Similarly, if the time spent in the page fault handler was more than a threshold of 5%, then 2MB page allocation is enabled but not 2MB page promotion, since there is little benefit in promoting the pages on which we had already paid the cost of page faults.

In order to estimate the impact of TLB misses on performance, we use *the fraction of L2 cache misses due to page table walks*. This assumes that TLB misses primarily degrade performance when a page table traversal causes an L2-cache miss (in that case, the miss is satisfied either from the L3 cache or from the DRAM, both of which are costly), and that the application's performance is dominated by L2 cache misses. Although this is a coarse approximation, it works well because applications that experience a lot of cache misses due to page table walks are those with large page tables. This implies that they have large memory footprints, and so they are memory-intensive. Therefore, it is safe to assume, for these applications, that variations in performance can be primarily explained by the number of L2 cache misses. Conversely, applications with a very small fraction of L2 cache misses resulting from page table walks are not memory-intensive, so for them the impact of TLB misses is negligible.

4 Evaluation

4.1 Performance evaluation

Figure 3 shows performance of Carrefour-LP and THP relative to Linux with 4K pages. We continue focusing only on the applications affected by NUMA issues; the remaining applications are presented for completeness in Figure 5. Figure 3 shows that Carrefour-LP:

- restores performance of applications that suffered under large pages and do not stand to benefit from them: CG.D, UA.B, UA.C,
- improves performance of applications that were expected to benefit from THP but did not (or did not benefit fully): SSCA and SPECjbb, both on machine A,
- does not significantly hurt performance of the applications where NUMA effects did not cause performance degradation under large pages and where no performance improvements from large pages were expected (the remaining applications).

We next provide the detailed analysis of Carrefour-LP. We analyze the contribution to performance improvements of its three components: Carrefour-2M, conservative and reactive. We demonstrate when and why it is sufficient to just use Carrefour-2M alone and explain how both conservative and reactive components contribute to the solution. The performance breakdown is shown in Figure 4.

Workloads other than CG.C, UA.B and UA.C are not affected by the hot-page effect and page-level false sharing, so in these cases Carrefour-LP performs similarly to Carrefour-2M alone. It is able to meet the performance of Carrefour-2M with minimal overhead (at most 3.7% on machine A and 2.1% on machine B).

Table 3 demonstrates that Carrefour-LP eliminates the hot-page effect and page-level false sharing and improves NUMA metrics where Carrefour-2M fails. For UA, the LAR drops from about 90% to roughly 60% under THP and remains at that low level under Carrefour-2M. Carrefour-LP is able to restore it almost to the previous level by dynamically splitting pages.

[5] Page promotion refers to dynamic consolidation of regular-sized pages into large pages. It is supported by the default Linux kernel. We set the frequency for page promotion checks to every 10ms.

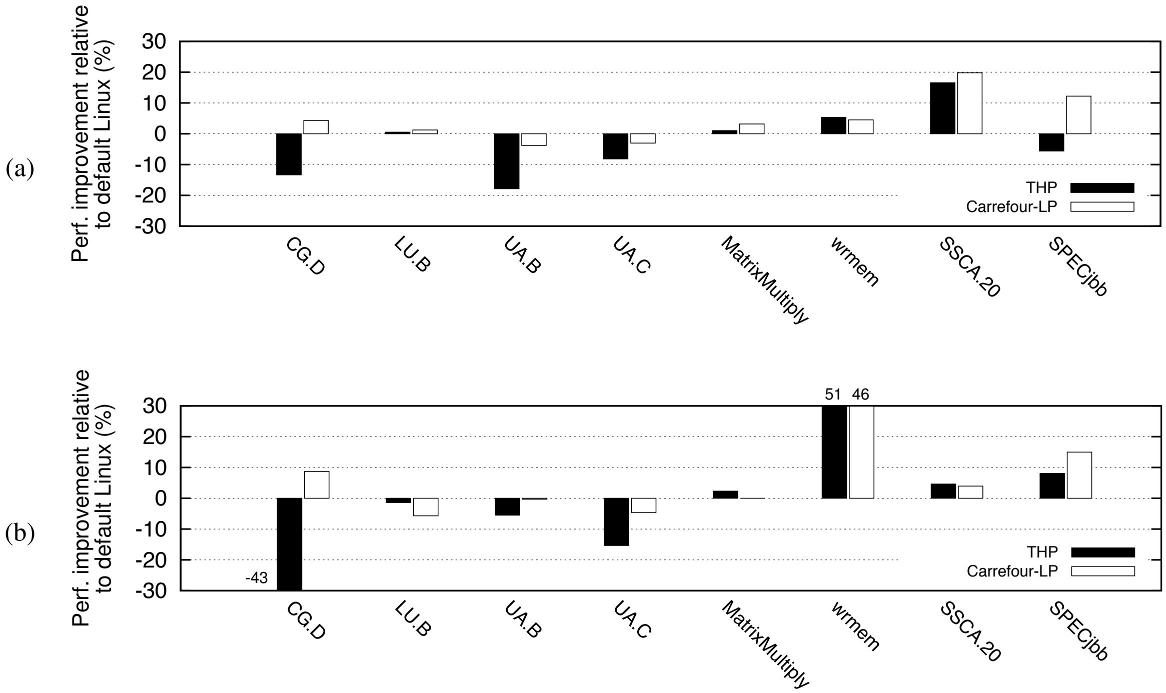


Figure 3: Performance improvement on a reduced set of applications of THP and Carrefour-LP over Linux, on (a) machine A and (b) machine B.

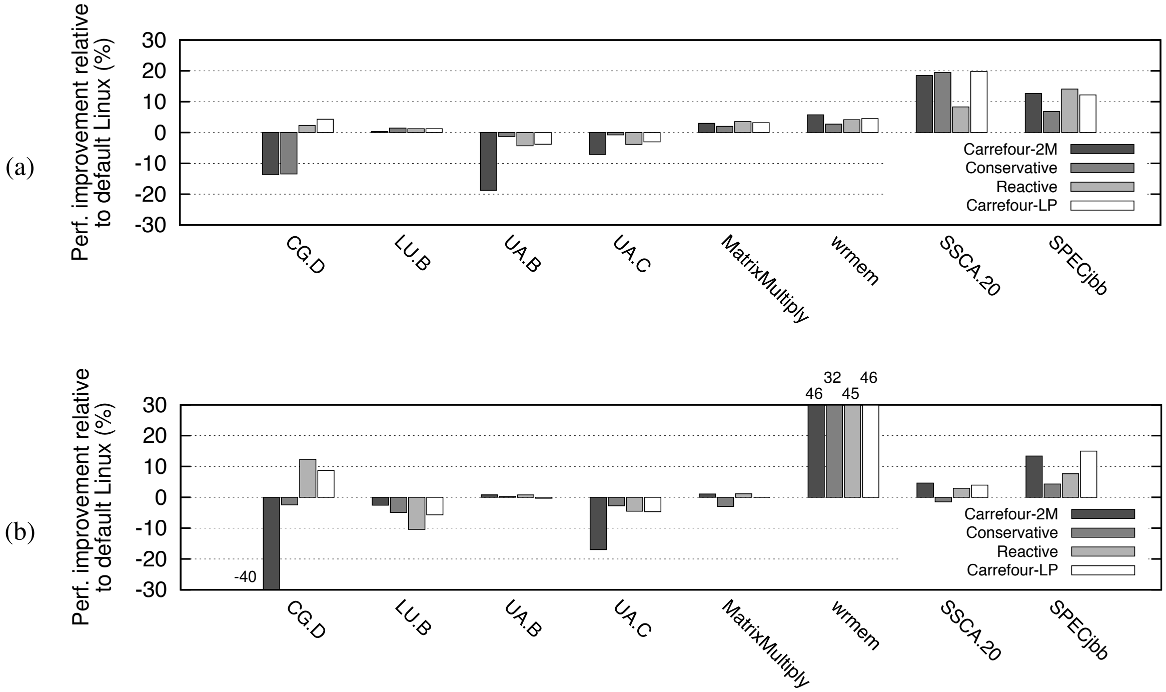


Figure 4: Performance improvement on a reduced set of applications of Carrefour-2M, the conservative component, the reactive component and Carrefour-LP over Linux with THP, on (a) machine A and (b) machine B.

For CG.D, enabling large pages disturbs the perfect memory-controller balance enjoyed under small pages. Carrefour-2M is unable to restore it, while Carrefour-LP restores it almost entirely.

We now analyze the importance of the two components in Carrefour-LP. Figure 4 presents the performance obtained when running Carrefour-2M alone (labeled as *Carrefour-2M*), Carrefour-2M with the reactive component designed for Carrefour-LP (labeled as *Reactive*), the original Carrefour runtime (working on 4kB pages) together with the conservative component (labeled as *Conservative*), and Carrefour-LP (labeled as *Carrefour-LP*). Figure 4 shows that in all cases, enabling the two components (as done in *Carrefour-LP*) is always the best choice (or close to the best). The conservative component alone does not solve the problem, because it begins with 4K pages. For SPECjbb, for example, it does not detect the need for large pages soon enough, so the performance is not as good as it could be. We similarly observed that using the conservative component alone hurts performance of many applications that were not included in this analysis (but shown in Figure 5) for the same reason: large pages were not enabled soon enough. These applications have an intense memory allocation phase at startup, which can benefit greatly from large pages due to fewer page faults, but the conservative component does not enable large pages soon enough.

Using the reactive component alone works well on some applications. For CG.D, it is able to detect the “hot page” and split it. Similarly, it is also able to split the falsely shared pages for UA.B and UA.C. However, on some applications, it fails to bring the maximum performance improvement that can be achieved with 2M pages (e.g. SSCA on machine A and SPECjbb on machine B). The reason is that the LAR is sometimes misestimated, and this results in 2M pages being split in applications that do not suffer from NUMA issues. For instance, on SSCA, the algorithm predicts a LAR of 59% if large pages were all split into 4k pages, whereas the actual LAR obtained after splitting is equal to 25%.

The problem is, in order to estimate the LAR under regular-sized pages given the data samples collected under large pages, we need to have enough samples on the constituent sub-pages. Unfortunately, we found it to be very difficult to gather enough samples; increasing the sampling rate results in unacceptably high overhead. A promising solution would be to use Lightweight Profiling (LWP). LWP is an extension of AMD processors that aims at providing the same level of details as IBS with less overhead. To reduce the overhead, LWP stores samples in a ring buffer and only interrupts the processor when the buffer is full. Unfortunately, on available AMD processors, LWP is only partially implemented: LWP samples only contain the instruction pointer of the sampled instruction and a timestamp. This information is not sufficient to predict LAR.

Because of these deficiencies in hardware profiling, the reactive component may make mistakes in deciding when to split large pages. This is where the conservative component comes to the rescue and re-creates the large pages when they are expected to help.

We conclude this section by explaining some performance results in Figure 5, which contains applications where THP did not create any NUMA issues. The key observation is that the overhead of Carrefour-LP does not significantly hurt these applications. Moreover, EP.C, SP.B and pca enjoy better (sometimes much better) performance with Carrefour-LP than with THP. That is because they had NUMA issues to begin with (which were not exacerbated by large pages), and so the Carrefour-2M component of the algorithm helped to address them.

4.2 Overhead assessment

Overhead in Carrefour-LP comes from collecting and storing IBS samples, computing the metrics based on these samples, migrating and splitting pages. Overall, the overhead of Carrefour-LP compared to the reactive approach is negligible: between 1% and 2% on all applications (on all machines) except CG (3.2%) and IS (2.1%) on machine B. Even on these two applications, the overhead is still within the standard deviation.

Compared to Carrefour-2M, the overhead is also small. The maximum overhead observed is 3.7% on machine A (SP.B) and 3.2% on machine B (LU.B), but on average it is below 2%.

Compared to Linux with 4k pages, Carrefour-LP has an overhead of less than 3%, except on FT, IS (machine A) and LU (machine B). This overhead is not specific to Carrefour-LP but is rather caused by Carrefour-2M, which spends too much time migrating large pages. Since our solution is built on top of Carrefour-2M, it also suffers from the same overhead.

4.3 Discussion

Our assessment of efficacy and downsides of Carrefour-LP is as follows.

The solution could be much improved if we had a more accurate way of estimating the LAR. Currently, with inaccurate estimates, the solution may split and migrate pages when there is no benefit to be gained, which is why Carrefour-LP degrades performance of LU by 3.5% compared to Carrefour-2M. We believe that the LAR could be predicted more accurately if we could collect more data samples without additional overhead. A complete implementation of LWP (i.e., if LWP provided

	Local Access ratio (%)				**Imbalance (%)**			
	Default Linux	*THP*	*Carr. 2M*	*Carr. LP*	*Default Linux*	*THP*	*Carr. 2M*	*Carr. LP*
CG.D (B)	40	36	38	39	**1**	**59**	**69**	**3**
UA.B (A)	**90**	**61**	**58**	**85**	9	15	17	10
UA.C (B)	**88**	**66**	**68**	**82**	14	12	9	14

Table 3: NUMA metrics for CG.D on machine B, UA.B on machine A, and UA.C on machine B.

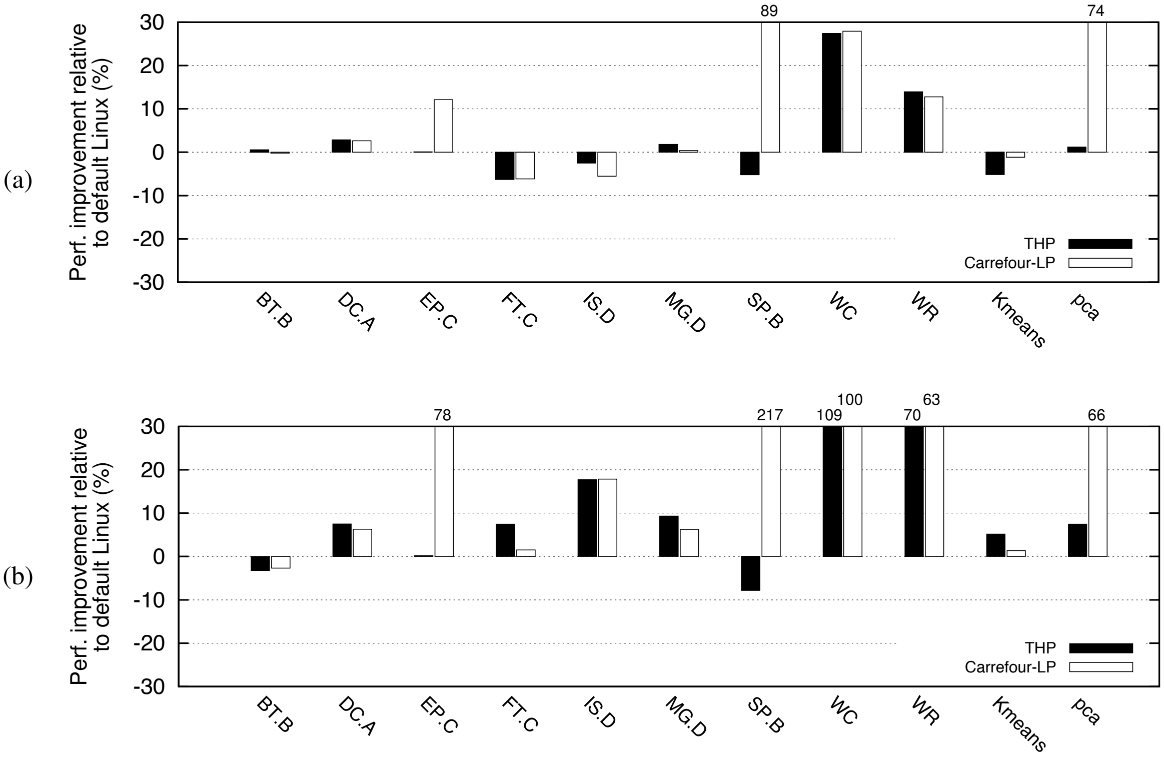


Figure 5: Performance improvement of THP and Carrefour-LP over Linux on applications whose NUMA metrics are **not** affected by THP, on (a) machine A and (b) machine B.

the same kind of samples as IBS) would solve this problem.

Our earlier implementation had scalability issues on the system with 64 cores. The reason was that the centralized data structure where we stored IBS samples had to be accessed and locked from multiple nodes. We addressed this problem by maintaining a data structure per node. The per-node structures are still accessed by multiple cores, so we may need to revisit this scaling issue on larger machines. Overall, the algorithm is likely to scale well because all work generated by an interrupt is performed independently on each node, so the number of nodes can grow without creating scalability bottlenecks.

Splitting pages did not create too much overhead, but the use of the page table lock for THP operations is clearly a scalability concern. Linux developers are working on finer grain locks at the time of the writing, so we hope that this problem will be avoided.

We did not observe many oscillations, where we go back and forth between splitting and enabling large pages. Overall, Carrefour-LP seems to be the more robust than the conservative and the reactive components used independently, because it naturally supports transient states and phase changes by continuously reexamining its decisions.

4.4 Very Large Pages

Although accessing the very large 1GB pages via libhugetlbfs proved challenging for most applications, we were able to enable them in SSCA and in streamcluster (an application from PARSEC)[6]. We immediately observed the hot-page and page-level false-sharing problems. With 1GB pages, lots of hot small pages were coalesced on a single NUMA node, and the performance dropped dramatically. For SSCA it degraded by 34%; for streamcluster by a factor of 4. Neither of these applications suffered performance degradation when 2M pages were used. Although preliminary, these data suggest a much more pervasive presence of NUMA issues when very large pages are used, and so Carrefour-LP will become even more important in the future.

5 Related Work

5.1 Large pages and TLB performance

Several studies have characterized the effect of TLB misses and large pages [2][10][15][14][7]. Battacharjee and Martonosi [2] specifically looked at the effect of TLB misses on multicore systems with multithreaded workloads. They found that some applications, such as Canneal from the PARSEC benchmark suite, spend up to 0.7 cycles per instruction on servicing D-TLB misses. Another study [10] showed performance improvements of up to 25% in the NAS benchmark suite due to using large pages. For large-scale HPC applications, Zhang et al. [15] found that large pages improve communication performance significantly.

Weisberg and Wiseman [14] used the SPEC CPU2000 benchmarks to evaluate the relationship between page size and the number of TLB misses. They argue that a 4KB page size is much too small for most applications, and conclude that a page size of 256KB and a 64-entry TLB is sufficient to drastically reduce the number of TLB misses.

Sudan et al. [12] motivate the need for small pages. They show that using 1KB pages allows optimizing the usage of the DRAM row-buffer, yielding substantial energy savings and decreasing the average latency of memory accesses.

All these works motivate the use of different page sizes, but none of them highlight or quantify the impact of NUMA on the performance obtained when using different page sizes.

5.2 Large page support and optimization

Many software systems have been designed that make large pages easier to use or more effective.

Navarro et al. [9] described an algorithm for operating system support of large pages that reduces fragmentation and does not require memory copies to create large pages. Using their algorithm, a page fault reserves a physical memory region of the size of a large page, but it initially only allocates and maps a small page. Subsequent page faults use the reserved space until it has been completely allocated, at which point the region is promoted to a large page. The algorithm does not attempt to optimize the placement of large pages.

Cascaval et al. [4] developed a model to predict the benefit of using large pages on individual data structures of applications, based on the predicted number of TLB misses and page faults. The predictions are computed using hardware counters throughout multiple runs of the application. The data structures that are predicted to benefit the most from large pages are backed by large pages. A similar method is described in [11], with the major difference being that large page promotions are performed at runtime.

Magee and Qasem [8] also devised a system for restricting the usage of large pages to applications that benefit the most from them. At compile-time, the working-set size is estimated through static analysis. If the estimated working-set size is greater than the coverage of the target CPU's TLB, then large pages are used.

A different approach is explored by Basu et al. [1]. Instead of managing the use of large pages at the OS level, they propose a hardware extension that allows applications to directly map memory segments. Addresses within directly mapped segments bypass the TLB and so translation is nearly free. The segments are conceptually similar to very large pages and provide similar benefits, but the authors do not analyze the potential NUMA effects which would be exacerbated by the large size of the segments.

In summary, previous works mostly focused on the limited availability of large pages and on reducing memory fragmentation. Several systems have been designed to ensure that applications that benefit from large pages actually use them, but no existing work has revealed and addressed the NUMA issues raised by large pages.

6 Conclusion

We demonstrated that using large pages can create or exacerbate NUMA issues like reduced locality or imbalance. We showed that these problems can be in some cases addressed by using a NUMA-aware page placement algorithm, but the latter stumbles upon two problems: the hot-page effect and page-level false sharing, which cannot be addressed via page migration. To address these problems, we implemented Carrefour-LP: large-page extensions to the NUMA-aware page placement algorithm Carrefour. Our results show that Carrefour-LP restores the performance when it was lost

[6] The PARSEC suite was not included in our study, because its applications did not experience performance differences under THP with 2M pages.

due to large pages and makes their benefits accessible to applications.

Solutions like Carrefour-LP will be even more important in the future, when very large pages (1GB in size) will be in widespread use.

References

[1] Basu, A., Gandhi, J., Chang, J., Hill, M. D., and Swift, M. M. Efficient virtual memory for big memory servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ACM, pp. 237–248.

[2] Bhattacharjee, A., and Martonosi, M. Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques* (2009), PACT '09, pp. 29–40.

[3] Boyd-Wickizer, S., Clements, A. T., Mao, Y., Pesterev, A., Kaashoek, M. F., Morris, R., and Zeldovich, N. An analysis of linux scalability to many cores.

[4] Cascaval, C., Duesterwald, E., Sweeney, P. F., and Wisniewski, R. W. Multiple page size modeling and optimization. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on* (2005), IEEE, pp. 339–349.

[5] Closson, K. Quantifying Hugepages Memory Savings with Oracle Database 11g, July 2009. http://kevinclosson.wordpress.com/2009/07/28/quantifying-hugepages-memory-savings-with-oracle-database-11g/.

[6] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quéma, V., and Roth, M. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems* (2013), ACM, pp. 381–394.

[7] Gorman, M., and Healy, P. Performance characteristics of explicit superpage support. In *Computer Architecture* (2012), Springer, pp. 293–310.

[8] Magee, J., and Qasem, A. A case for compiler-driven superpage allocation. In *Proceedings of the 47th Annual Southeast Regional Conference* (2009), ACM, p. 82.

[9] Navarro, J., Iyer, S., Druschel, P., and Cox, A. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation* (2002), OSDI '02, pp. 89–104.

[10] Noronha, R., and Panda, D. K. Improving scalability of openmp applications on multi-core systems using large page support. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* (2007), IEEE, pp. 1–8.

[11] Romer, T. H., Ohlrich, W. H., Karlin, A. R., and Bershad, B. N. Reducing tlb and memory overhead using online superpage promotion. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on* (1995), IEEE, pp. 176–187.

[12] Sudan, K., Chatterjee, N., Nellans, D., Awasthi, M., Balasubramonian, R., and Davis, A. Micro-pages: increasing dram efficiency with locality-aware data placement. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems* (2010).

[13] Talluri, M., Kong, S., Hill, M. D., and Patterson, D. A. Tradeoffs in supporting two page sizes. In *Computer Architecture, 1992. Proceedings., 19th Annual International Symposium on* (1992).

[14] Weisberg, P., and Wiseman, Y. Using 4kb page size for virtual memory is obsolete. In *Information Reuse & Integration, 2009. IRI'09. IEEE International Conference on* (2009), IEEE, pp. 262–265.

[15] Zhang, P., Li, B., Huo, Z., and Meng, D. Evaluating the effect of huge page on large scale applications. In *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on* (2009), IEEE, pp. 74–81.

Efficient Tracing of Cold Code via Bias-Free Sampling

Baris Kasikci [1*], Thomas Ball [2†], George Candea [1‡], John Erickson [2§], and Madanlal Musuvathi [2¶]

[1]School of Computer and Communication Sciences, EPFL
[2]Microsoft

Abstract

Bugs often lurk in code that is infrequently executed (i.e., cold code), so testing and debugging requires tracing such code. Alas, the location of cold code is generally not known a priori and, by definition, cold code is elusive during execution. Thus, programs either incur unnecessary runtime overhead to "catch" cold code, or they must employ sampling, in which case many executions are required to sample the cold code even once.

We introduce a technique called *bias-free sampling* (BfS), in which the machine instructions of a dynamic execution are sampled independently of their execution frequency by using breakpoints. The BfS overhead is therefore independent of a program's runtime behavior and is fully predictable: it is merely a function of program size. BfS operates directly on binaries.

We present the theory and implementation of BfS for both managed and unmanaged code, as well as both kernel and user mode. We ran BfS on a total of 679 programs (all Windows system binaries, Z3, SPECint suite, and on several C# benchmarks), and BfS incurred performance overheads of just 1–6%.

1 Introduction

Monitoring a program's control-flow is a fundamental way to gain insight into program behavior [5]. At one extreme, we can record a bit per basic block that measures whether or not a block executed over an entire execution (coverage) [29]. At another extreme, we can record the dynamic sequence of basic blocks executed (tracing) [28]. In between these two extremes there is a wide range of monitoring strategies that trade off runtime overhead for precision. For example, record-replay systems [12, 15] that record most execution events in a program incur a large overhead, whereas sampling strategies that collect fewer runtime events for both profiling and tracing [16] incur less overhead.

In testing and debugging, there is a need to sample infrequently executed (i.e., cold) instructions at runtime, because bugs often lurk in cold code [9, 23]. However, we don't know a priori which basic blocks will be cold vs. hot at runtime, therefore we cannot instrument just the cold ones. To make matters worse, traditional temporal sampling techniques [21, 24] that trade off sampling rate for sampling coverage can miss cold instructions when the sampling rate is low, requiring many executions to gain acceptable coverage. As a result, developers do not have effective and efficient tools for sampling cold code.

In this paper, we present a non-temporal approach to sampling that we call *bias-free sampling* (BfS). BfS is guaranteed to sample cold instructions without oversampling hot instructions, thereby reducing the overhead typically associated with temporal sampling.

The basic idea is to sample any instruction of interest *the next time* it executes and *without* imposing any overhead on any other instructions in the program.

We do this using code breakpoints (a facility present in all modern CPUs) dynamically. We created *lightweight code breakpoint* (LCB) monitors for both the kernel and user mode of Windows for both native (with direct support in the kernel) and managed applications (with a user-space monitor) on both Intel and ARM architectures.

To ensure that none of the cold instructions are missed, the bias-free sampler inserts a breakpoint at *every* basic block in the program, both at the beginning of the program execution and periodically during the execution. This ensures at least one sample per period of every cold instruction. We also show how to sample without bias hot instructions independently of their execution frequency at a low rate.

Devising an efficient solution that works well in prac-

[*] baris.kasikci@epfl.ch
[†] tball@microsoft.com
[‡] george.candea@epfl.ch
[§] jerick@microsoft.com
[¶] madanm@microsoft.com

tice on a large set of programs requires solving multiple challenges: (a) processing a large number of breakpoints, in the worst case simultaneously on every instruction in the program (existing debugging frameworks are unable to handle such high volumes because their design is not optimized for a large number of breakpoints that must be processed quickly); (b) handling breakpoints correctly in the presence of a managed code interpreter and JIT optimizations (managed code gets optimized during execution, therefore it cannot be handled the same way as native code); and (c) preserving the correct semantics of programs and associated services, such as debuggers.

A particular instance of LCB that we built is the lightweight code coverage (LCC) tool. We have successfully run LCC at scale to simultaneously measure code coverage on all processes and kernel drivers in a standard Windows 8 machine with imperceptible overheads. We also have extended LCC with the ability to record periodic code coverage logs. LCC is now being used internally at Microsoft to measure code coverage.

Using breakpoints overcomes many of the pitfalls of code instrumentation. CPU support for breakpoints allows setting (a) a breakpoint on *any* instruction, (b) an arbitrary number of breakpoints, and (c) setting or clearing a breakpoint without synchronizing with other threads (with the exception of managed code) that could potentially execute the same instruction.

The contributions and organization of this paper are:

- We analyze and dissect common approaches to cold code monitoring, showing that there is need for improvement (§2);
- We present our BfS design (§3) and its efficient and comprehensive implementation using breakpoints for both the kernel and user mode of Windows for both native and managed applications (§4);
- We show on a total of 679 programs that with our implementation of LCB, our coverage tool LCC, which places a breakpoint on every basic block in an executable and removes it when fired, has an overhead of 1-2% on a variety of native C benchmarks and an overhead of 1-6% on a variety of managed C# benchmarks (§5);
- We show how to use periodic BfS to extend LCC to quickly build interprocedural traces with overheads in the range of 3-6% (§6).

§7 discusses related work and §8 concludes with a discussion of applications for BfS.

2 From Rewriting to Bias-Free Sampling

In this section, we provide background on the approaches used to monitor program behavior, and outline the conceptual path that leads to our proposed technique.

2.1 Program Rewriting

A traditional approach to monitoring program behavior is *static program rewriting* as done by Gcov [13], which takes as input an executable E and outputs a new executable E' that is functionally the same as E except that it monitors the behavior of E. At Microsoft, many such monitoring tools have been built on top of the Vulcan binary rewriting framework [27], such as the code coverage tool `bbcover`. Vulcan provides a number of program abstractions, such as the program control-flow graph, and the tool user can leverage these abstractions to then use the Vulcan APIs to add instructions at specific points in the binary. Vulcan ensures that the branches of the program are adjusted to reflect this addition of code.

Another approach to monitoring is *dynamic program rewriting*, as done by DynInst [7] and Pin [22], as well as Microsoft's Nirvana and iDNA framework [6]. Many of the tools built with rewriting-based approaches, both static and dynamic, use "always-on" instrumentation (they keep the dynamically-added instrumentation until the program terminates), even when for goals that should be much less demanding, like measuring code coverage.

2.2 Efficient Sampling

Static or dynamic program rewriting approaches that are always-on incur prohibitive overheads, and they cannot sample cold code in a bias-free manner.

In 2001, Arnold et al. introduced a framework for reducing the cost of instrumented code that combines instrumentation and counter-based sampling of loops [24]. In this approach, there are two copies of each procedure: The "counting" version of the procedure increments a counter on procedure entry and a counter for each loop back edge, ensuring that there is no unbounded portion of execution without some counter being incremented. When a user-specified limit is reached, control transfers from the counting version to a more heavily instrumented version of the procedure, which (after recording the sample) transfers control via the loop back edges back to the counting version. In this way, the technique can record more detailed information about acyclic intraprocedural paths on a periodic basis.

Hirzel et al. extended this method to reduce overhead further and to trace interprocedural paths [17]. They implemented "bursty tracing" using Vulcan, and report runtime overheads in the range of 3-18%. In further work [16] they sample code at a rate inversely proportional to its frequency, so that less frequently executed code is sampled more often. This approach is based on the premise that bugs reside mainly on cold paths.

Around the same time, Liblit et al. [21] proposed "Sampling the Bernoulli Way" in their paper on what later was termed "cooperative bug isolation." The motivation for their approach was that classic sampling for measuring program performance "searches for the 'elephant in the haystack': it looks for the biggest consumers of time" [21]. In contrast, the goal is to look for needles (bugs) that may occur rarely, and the sampling rates may be very low to maintain client performance. This leads to the requirement that the sampling be statistically fair, so that the reported frequencies of rare events be reliable. The essence of their approach is to perform fair and uniform sampling from a dynamic sequence of events. To obtain sufficient samples of rare events, their approach relies on collecting a large number of executions.

2.3 Bias-Free Code Sampling

There's a fundamental tension between the desire to look for needles in a haystack (cold code), the use of bursty tracing, and Bernoulli sampling to achieve efficiency. Bursty tracing can trace cold code at a high cost; sampling is efficient, but it requires many runs before cold paths are sampled, and thus may incur a large overhead.

Consider the simple example of a hot loop containing an `if-then-else` statement where the `else` branch is very infrequently executed compared to the loop head—say the `else` branch executes once every million iterations of the loop. The desire to keep the sampling rate low for efficiency means it's unlikely that Bernoulli sampling or the bursty tracing approach will hit upon the one execution of the `else` branch in a million iterations.

Furthermore, we generally do not know a priori which code blocks will be cold during the execution of interest. Thus, we need a way to sample all code but not let the different frequencies of execution of the different code blocks influence the runtime performance overhead. In other words, the sampling rate of a code block should be (mostly) independent of how often it is executed. We say "mostly" because there still is a dependency: the block must be executed at least once for it to be sampled.

The basic idea behind our approach is (using the example above) that placing a breakpoint on the first instruction in the `else` branch guarantees that we will sample the next (albeit rare) execution of the `else` branch with no cost for the many loop iterations before that point. By refreshing this breakpoint periodically, we can obtain several samples of this rare event.

Looking at it from the other side, Bernoulli sampling gives equal likelihood that any of the million loop iterations of the loop's execution will be sampled. This may be fair to all the loop iterations, but it doesn't help identify the cold code. Cooperative bug isolation makes up for the fact that a single execution may not uncover cold code by the law of large numbers (of executions) to increase the confidence that a rare event will be sampled.

> Bias-free code sampling is a way to sample cold events just as efficiently as Bernoulli sampling with far fewer executions.

3 Design

The core idea of BfS is to use breakpoints to: (a) sample cold instructions that execute only a few times during an execution, without over-sampling hot instructions; (b) sample the remaining instructions independently of their execution frequency. Algorithm 1 presents the BfS algorithm. We discuss the algorithm in its full generality before discussing particular instantiations.

3.1 Inputs

The algorithm takes as input three parameters. The parameter `K` ensures that the first `K` executions of any instruction are always sampled. Assuming a nonzero `K`, this ensures that rare instructions, such as those in exception paths, are always sampled when executed.

The second parameter `P` is the sampling distribution of the instructions. For instance, a memory leak detector [16] might only chose to sample memory access instructions, and accordingly `P` will indicate a zero probability for non-memory-accesses. Similarly, a data-race detector [18] might only choose memory accesses that are not statically provable as data-race-free. Among the instructions with non-zero probability, `P` might either dictate a uniform sampling, or bias towards some instructions, based on application needs. For instance, additional runtime profile information could be used to increase the bias towards hot instructions or towards instructions that are likely to be buggy.

The final parameter `R` determines the desired sampling rate, i.e., the number of samples generated per second. This indirectly determines the overhead of the algorithm. In the special case when `R` is infinity, the algorithm periodically refreshes breakpoints on all instructions selected according to `P`.

3.2 Cold Instruction Sampling

The algorithm maintains a map `BPCount` that determines the number of logical breakpoints set at a particular instruction. The algorithm ensures that a breakpoint is set at a particular instruction whenever its `BPCount` is greater than zero. When a breakpoint fires, this count is decremented and the breakpoint is removed only when this count is zero. Setting all entries of this array to `K` ensures that the first `K` executions of the instructions with nonzero probability in `P` are sampled.

Algorithm 1: Bias-free Sampling Algorithm

```
Input: int K, Dist P, int R
// BPCount[pc] > 0 implies pc has a breakpoint
$Map < PC, int > BPCount$
$Map < PC, int > SampleCount$
$Set < PC > FreqInst$
function Init
  For all pc with nonzero probability in P
    $BPCount[pc] = K$
function OnBreakpoint(pc)
  $BPCount[pc] - -$
  $SampleCount[pc] + +$
  $SampleInstruction(pc)$
  if $SampleCount[pc] >= K$ then
    FreqInst.Add(pc)
  if $SampleCount[pc] > K$ then
    $ChooseRandomInst()$
function Periodically()
  $hitNum = NumBPInLastPeriod()$
  if $R$ is infinity then
    $BPCount[pc] + +$ for all pc in FreqInst
    return
  while $hitNum + + < R * Period$ do
    ChoseRandomInst()
function ChooseRandomInst()
  $pc = Choose(P, FreqInst)$
  $BPCount[pc] + +$
  $FreqInst.Remove(pc)$
```

At one extreme,With K 1, P choosing only the first instruction in every basic block, and R as 0, we obtain an efficient mechanism for code coverage, described as LCC in Section 5. On the other extreme, when K is set to infinity, one gets full execution tracing.

The algorithm maintains FreqInst, a set of instructions that have executed K or more times. Periodically, the algorithm adds breakpoints to instructions selected from this set based on the distribution P. When one such breakpoint fires, another breakpoint is inserted on an instruction chosen from this set, again based on P. One can consider this as a single logical breakpoint moving from the sampled instruction to a new instruction. To maintain the number of pending logical breakpoints, the algorithm uses the BPCount map to distinguish the initial K breakpoints from new breakpoints.

3.3 Bias-Free Sampling

Perhaps surprisingly, the algorithm described above is sufficient to sample instructions from FreqInst based on P irrespective of whether these instructions are hot or cold. Since an instruction is sampled only when a breakpoint fires and these breakpoints are inserted based on P, we meet the desired sampling distribution [18].

However, a single breakpoint set at a cold instruction may take a long time to fire. This can arbitrarily reduce the sampling rate achieved by this logical breakpoint. In the worst case, a breakpoint set in dead code will reduce the sampling rate to zero.

The algorithm has several mechanisms to avoid this pitfall and maintain an acceptable sampling rate. First, the algorithm starts by setting K logical breakpoints at every instruction. This helps in identifying only those instructions that have executed a few times. In particular, dead code will not be added to FreqInst. Second, once a breakpoint is set at an instruction, it will be removed from FreqInst till it fires (at which point it is added back to FreqInst). This mechanism automatically prunes cold instructions from the set to periodically replenish the number of logical breakpoints. This is similar to DataCollider [18], however, rather than maintaining a constant number of pending logical breakpoints, our algorithm increases the number of logical breakpoints in every period that has a lower number of breakpoint firings than expected by the sampling rate. As these logical breakpoints get “stuck” on cold instructions, the continuous replenishing helps maintain the sampling rate.

4 Implementation

Now, we describe the implementation of LCB in detail. We start by reviewing hardware and operating system support for breakpoints.

4.1 Breakpoint Mechanism

Modern hardware contain a special breakpoint instruction that tells the processor to trap into the operating system. For instance, the x86 architecture provides an `int 3` instruction for this purpose. To set a breakpoint on an instruction, one overwrites the instruction with the breakpoint instruction. The breakpoint instruction is no larger than other instructions in the ISA (in x86, the breakpoint instruction is a single byte), making it possible to set a breakpoint without overwriting other instructions in the binary. When a breakpoint fires, the operating system forwards the interrupt to the process or to the debugger if one is attached. Processing the breakpoint involves removing the breakpoint by writing back the original instruction at the instruction pointer and resuming the program. The breakpoint instruction is designed so that setting and removing a breakpoint can be done atomically in the presence of other threads that might be executing the same instructions. For example, in architectures (such as ARM) that support two-byte breakpoint instructions, all instructions are always two-byte aligned.

4.2 Kernel Support

One of the key goals of LCB is to provide a general capability to set and remove a large number of breakpoints as efficiently as possible. Equally important is to do so without changing the semantics of the monitored programs and associated services such as debuggers. LCB relies on kernel processing for efficient and transparent processing of breakpoints. While most of the functionality of LCB can be implemented as a kernel driver that is loaded early in the boot sequence, we relied on some modifications to the Windows kernel. Another advantage of kernel support is that we can use LCB to sample kernel-mode drivers as well.

4.3 Efficient Processing of Breakpoints

4.3.1 Bypassing the Debugger

When a breakpoint fires, the default behavior of the kernel is to notify the debugger (if attached) or send the interrupt to the process. LCB driver registers itself as a debugger so that it gets a first chance to process the interrupt. Bypassing the regular debugger is crucial for efficiency, as debuggers do not handle well frequent firing of any breakpoints. The LCB driver forwards the interrupt to the debugger or to the process if the breakpoint is not one inserted by LCB.

4.3.2 Handling Shared Modules

Another key design decision of LCB is how to handle shared modules. The code section of modules that are frequently loaded by many processes, such as the C libraries, are loaded in memory once and shared across many processes through appropriate virtual memory mapping. Setting a breakpoint at an instruction in such a shared module can be implemented in one of two ways. The first option is to make the breakpoint common to *all* processes. Thereby, the sampling of the instruction is triggered when any of the processes executes the instruction. Another option is to create a per-process copy of the memory page containing the instruction, causing the loss of memory savings achieved by sharing the module.

The current design of LCB uses the first option for efficiency. In many of our usage scenarios, LCB is turned on for many processes, and the memory bloat that would occur as a result of choosing the second option is unacceptable (as LCB sets breakpoints on all code pages). Moreover, this allows us to extend LCB-based sampling for multiprocess programs. For instance, when measuring code coverage, any of the processes executing a particular C library function is sufficient to cover that function.

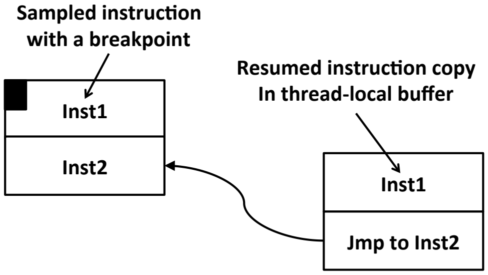


Figure 1: Implementation of multi-shot breakpoints.

4.3.3 Handling Multi-Shot Breakpoints

The functionality LCB provides may require resuming the currently sampled instruction *without* removing the breakpoint. Such multi-shot breakpoints are required, for instance, for sampling the first K executions of a basic block. This goes against the default processing of breakpoints, where the breakpoint needs to be removed before resuming the sampled instruction. Once LCB has resumed the execution, it would not get back control unless another breakpoint fires. In the interim, the sampled instruction could have executed many times.

Another option is to use the single-stepping capability of modern architectures. For instance, setting the Trap Flag in the EFLAGS register causes the x86 processor to generate an interrupt after executing a single instruction. Debuggers use this facility to single-step an instruction after removing its breakpoint and on the subsequent interrupt (caused by single-stepping) set the breakpoint on that instruction again. This is safe as debuggers usually block all other threads during this process, however, this generally has unacceptable overhead.

To handle multi-shot breakpoints in native code, LCB creates a copy of the currently sampled instruction in a thread-local buffer, as shown in Figure 1. Immediately after the copy, LCB inserts a jump instruction to transfer control to the instruction after the sampled instruction. When returning from the breakpoint handler, LCB sets the current instruction pointer to the copy of the instruction. This allows the current thread to resume execution without removing the breakpoint. The jump after the copy ensures that control returns to the original program. Note that, this design works even if the sampled instruction is a jump or branch instruction, in which cases the jump instruction of the copy is not executed.

When creating a copy of the instruction, one has to carefully handle instructions that refer to the instruction pointer. For instance, *relative* jump instructions calculate their destination based on the current instruction pointer. Such instructions need to be appropriately modified to retain their semantics when creating a copy. While LCB handles many common cases, it defaults to single-stepping (with all other threads blocked) for other instructions that refer to the instruction pointer.

The instruction copy in the thread-local buffer is reclaimed by the thread when it ensures that its current instruction pointer and the return values in its stack trace do not point to the copy. For kernel-mode drivers, LCB allocates a processor-local buffer, rather than a thread-local one. This buffer is shared by all contexts that execute on a particular process, including interrupt handlers.

4.4 BfS for Managed Code

Supporting managed code in LCB (such as code written in .NET languages and encoded into the Common Intermediate Language (CIL)) required overcoming several challenges: integration with the Common Language Runtime [2], making sure that the just-in-time (JIT) optimizations do not remove certain breakpoints, and finding and fixing issues in CLR that prohibited setting a large number of breakpoints. In this section, we detail how we overcame these challenges[1].

Initially, we attempted to place breakpoints on every basic block without going through the CLR debugging APIs. However, this did not work, because CLR introspects the managed binary during JITing, and if it finds that the binary has been modified (in this case to include a breakpoint per basic block), it throws an exception and causes the program to crash.

Consequently, we used the CLR debugging APIs to support managed programs in LCB. To do this, we implemented a special debugger within LCB that intercepts the load of each managed module when a program is run and places a breakpoint in each of the program's basic blocks. This debugger's core responsibility is to place breakpoints and track their firing. A program need not be launched using this debugger for LCB to be operational: LCB can be automatically attached to a program at load time.

The second challenge was that the CLR JIT optimizations were modifying the programs by eliminating some basic blocks (e.g., through dead-code elimination) or by moving them around (e.g., through loop-invariant code motion), causing the correspondence between the removed breakpoints and source code to be lost.

To overcome this challenge, we added an option to LCB to disable JIT optimizations and obtain perfect correspondence between the source code and the basic blocks. We are looking into recovering the lost correspondence through program analysis as part of future work, thereby not forcing users of LCB to disable JIT optimizations.

4.5 Transparent Breakpoint Processing

For a facility that is commonly used, such as breakpoints, one would not expect the use of breakpoints to change the semantics of programs. While this is generally true, we had to handle several corner cases in order to apply LCB to a large number of programs.

4.5.1 Code Page Permissions

Setting a breakpoint requires write permission to modify the code pages. However, for security purposes, all code pages in Windows are read-only. A straightforward approach is to change the permission to enable writes, then set/clear the breakpoint, and then reset the permission to *readonly*. However, this leaves a window in which another (misbehaving) thread could potentially write to the code page. Under such conditions, the original program would have received an access violation exception while the same program running with LCB would not.

To avoid this, LCB creates an alternate virtual mapping to the same code page with write permissions and uses this mapping to set and clear breakpoints. This mapping is created at a random virtual address to reduce the chances of a wild memory access matching the address. The virtual mapping is cached to amortize the cost of creating the mapping across multiple breakpoints—due to code locality, breakpoints in the same page are likely to fire together.

When sampling kernel-mode drivers, LCB sometimes has the need to process breakpoints at interrupt levels during which it is unable to call virtual-memory-related functions to create/tear down virtual mappings. In such scenarios, LCB uses the copy mechanism for dealing with multi-shot breakpoints described above (§ 4.3.3) to temporarily resume execution without removing a breakpoint. At the same time, LCB queues a deferred procedure call that is later invoked at a lower interrupt level to remove the breakpoint.

Finally, LCB does not set or clear breakpoints on code pages that are writable in order to avoid conflicts with self-generated code.

4.5.2 Making Breakpoints Invisible to the Debugger

Many programs with LCB enabled run with a debugger attached. As described above, LCB hides its breakpoints from the debugger by processing them before the debugger. However, debuggers need to read the code pages,

[1] In the process of implementing LCB for managed programs, we discovered and fixed performance bottlenecks and bugs in the CLR. CLR debugging APIs had such issues, because they were not built to be used by a client such as LCB that places a breakpoint in each basic block of a program. The first bug we fixed was a performance issue that caused threads to unnecessarily stall while LCB was removing a breakpoint, due to an incorrect spinlock implementation. The second bug was a subtle correctness issue that occurred only when the number of breakpoints was above 10,000, and JIT optimizations were enabled. We also fixed this issue that was causing the CLR to crash.

say in order to disassemble code to display to the user. LCB traps such read requests and provides an alternate view with all its breakpoints removed.

5 LCC Evaluation

In this section, we measure the cost of placing "one-shot" breakpoints on every basic block in an executable using LCB monitors in order to measure code coverage. The resulting code coverage tool is called LCC. LCC represents the leanest instance of LCBẆe first perform a case study on the Z3 automated theorem prover [10] (§5.1), followed by a broader investigation on the SPEC 2006 CPU integer benchmarks (§5.2), then three managed benchmarks from the CLR performance benchmarks (§5.3), and a large scale evaluation on Windows binaries (§5.4).

The code coverage evaluations were performed on an HP Desktop with a 4-core Intel Xeon W3520 and 8 GB of RAM running Windows 8. In our study, we consider three configurations for each application: no code coverage (*base*), the application statically rewritten by the `bbcover` tool (*bbcover*), and the application breakpoint-instrumented by LCC (*lcc*). In order to make the comparison between the tools as fair possible, we use the same basic blocks for LCC breakpoints as identified by the Vulcan framework for the `bbcover` tool. We instruct LCC to insert a breakpoint at the address of the first instruction in each basic block. On the firing of a breakpoint, a bit (in a bitvector) is set to indicate that the basic block has been covered.

5.1 Z3

Z3 is an automated theorem prover written in C++ consisting of 439,927 basic blocks (as measured by Vulcan). Z3 is computationally and memory intensive, having a SAT solver at its core, which is solving an NP-complete problem. We run Z3 on a set of 66 input files that take Z3 anywhere from under 1 second to 150 seconds to process (and many points in between). Each file contains a logic formula over the theory of bit vectors (generated automatically by the SAGE tool [14]) that Z3 attempts to prove satisfiable or unsatisfiable. Z3 reads the input file, performs its computation, and outputs "sat" or "unsat". We test the 64-bit version of the Release build of Z3. For each test file, we run each configuration five times.[2]

We added timers to LCC to measure the cost of setting breakpoints, which comes to about 100 milliseconds to set all 439,927 breakpoints.

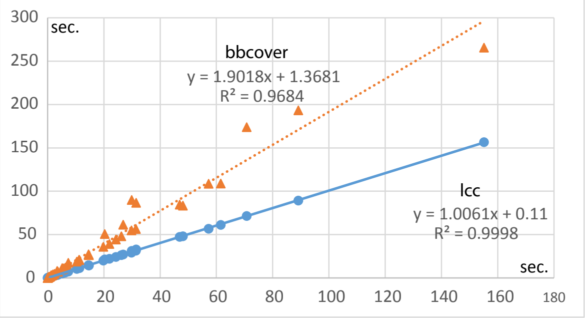


Figure 2: Plot comparing the absolute run-times of coverage tools `bbcover` (triangles, upper line) and LCC (circles, lower line) on the Z3 program (y-axis) against the base configuration (x-axis). Both times are seconds.

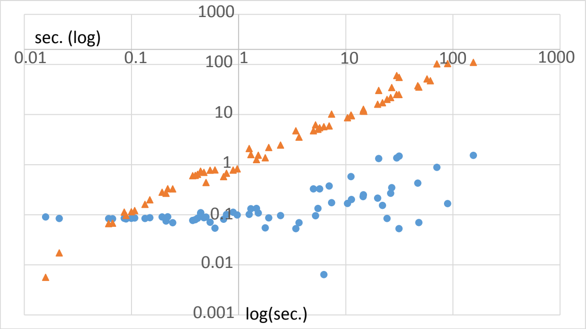


Figure 3: Log-log plot comparing the overhead of `bbcover` (orange triangles) and LCC (blue circles), in seconds over base (y-axis), as a function of base (x-axis).

Figure 2 plots the absolute run time of each test *t* for the base configuration (the median of 5 runs) against each of the two code coverage configurations and shows the best linear fit for each configuration. We see that the overhead for LCC is less than 1%, with much less perturbation than the overhead of `bbcover`, while the overhead for `bbcover` is around 90% and has outliers.

We would expect that the overhead for LCC be a small constant, independent of the running time of the base execution. In the log-log plot of Figure 3, the x-axis is the run-time in seconds of the base configuration on a test *t*, while the y-axis represents the overhead (in seconds) of each of the code coverage configurations (over the base configuration) on the same test *t*. We see the expected linear relationship of the cost of code coverage with respect to execution time for `bbcover`. The plot for LCC shows that the overhead for LCC appears to increase slightly with the base time, although its overhead never exceeds 1.5 seconds.

Figure 4 shows the number of basic blocks (y-axis) covered as a function of run-time (x-axis, log scale). The first thing to notice is that most of the tests cover somewhere between 17,000 and 29,000 basic blocks, a

[2] We validated that the output of Z3 is the same when run under each code coverage configuration as in the base run and that the coverage computed by LCC is the same as that computed by `bbcover`.

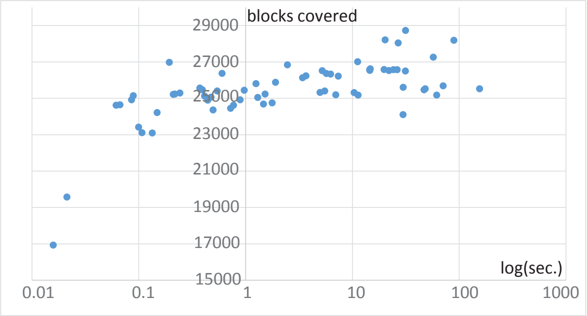


Figure 4: Run-time of base configuration (x-axis, in seconds on log scale) versus number of basic blocks covered, for each of the 66 tests.

small fraction of all the basic blocks in Z3. This is not a surprise, as the 66 tests were selected from a suite that exercises just a part of Z3 (the bit vector theory and SAT solver). The two tests that cover less than 21,000 blocks also have the shortest runtimes. Block coverage increases slightly as execution time increases, correlated with the observed increase in runtime overhead for LCC.

5.2 SPEC CPU2006 Integer Benchmarks

To understand the cost of code coverage on a wider set of programs, we integrated both bbcover and LCC into the SPEC CPU2006 Monitoring Facility and performed experiments on the SPEC 2006 CPU Integer benchmarks (except for 462.libquantum and 483.xalancbmk, which did not compile successfully on Windows 8).

Table 1 presents the results of the experiments, with one row per benchmark. We ran each benchmark for five iterations using base tuning. The second and third column show the number of basic blocks in a benchmark and the number of tests for that benchmark (each iteration runs all tests once and sums the results). We call out the number of tests because each test is a separate execution of the benchmark, which starts collection of code coverage afresh. Thus, for example, the 403.gcc benchmark has 9 tests and so will result in setting breakpoints 9 times on all 198719 blocks (for one iteration). The columns labeled *base*, *lcc*, and *bbcover* are the median times reported by the runspec script (in seconds) of the five runs, for each configuration, respectively, as well as the standard deviation. The overhead of the *lcc* and *bbcover* configurations to the base configuration is reported in the remaining two columns.

The overhead of bbcover ranges from a low of 18.67% (429.mcf) to a high of 176.22% (400.perlbench). In general, the slowdown varies quite a bit depending on the benchmark. Our experience with static instrumentation is that the number of the frequently executed basic blocks in the executable is the main determiner of overhead. The overhead of LCC, on the other hand, ranges from 1.4% to 2.18%, showing that LCC achieves low overhead across a range of benchmarks, despite the high cost of breakpoints.

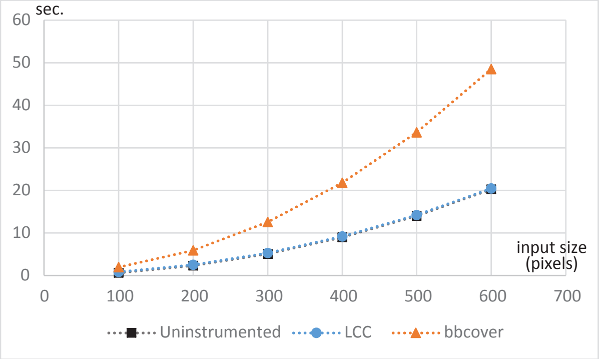


Figure 5: Plot comparing absolute runtimes in seconds (y-axis) of bbcover (triangles), LCC (circles), and uninstrumented execution (squares) on the RayTracer program as a function of input size (x-axis).

5.3 Managed Code

We evaluated LCB's managed code support using three programs used internally at Microsoft for CLR performance benchmarking: RayTracer, a program that performs ray tracing; BizTalk, a server application used in business process automation; and ClientSimulator, a web client simulation program. We measured the uninstrumented runtimes and coverage measurement overheads for bbcover and LCC. All results are averages of five runs.

In Figure 5, we vary the size of the input object RayTracer processes from 100 to 600 pixels to see how the overhead changes with input size. The y-axis shows the absolute runtime. The runtime overhead of LCC is a steady 0.2 seconds corresponding to a maximum of 6% overhead irrespective of input size, whereas the runtime overhead of bbcover is proportional to the runtime of RayTracer with a maximum absolute time of 28 seconds and a maximum overhead factor of 3×.

Similar to the native Z3 binary, this experiment shows that for the managed RayTracer binary, LCC's overhead is less than that of bbcover and it is independent of the program's runtime behavior.

For BizTalk and ClientSimulator, we used standard workloads of the benchmarks. For Biztalk, LCC incurs 1.1% runtime overhead versus bbcover's 2.0% overhead; for ClientSimulator, LCC incurs 5.8% runtime overhead versus bbcover's 34.7% overhead.

RayTracer has several loops that execute many times, therefore, for this case, the runtime overhead of bbcover (which instruments the code) is two orders of magni-

Benchmark	num. of blocks	num. of tests	*base* (sec.)	std. dev.	*lcc* (sec.)	std. dev.	*bbcover* (sec.)	std. dev.	*lcc* overhead	*bbcover* overhead
400.perlbench	68224	3	473.71	0.98	481.98	1.33	1308.49	12.31	1.75%	176.22%
401.bzip2	6667	6	575.02	0.77	584.31	2.57	1108.96	5.73	1.62%	92.86%
403.gcc	198719	9	402.27	0.81	410.55	2.75	765.55	1.32	2.06%	90.31%
429.mcf	5363	1	366.49	0.66	373.00	5.50	434.93	0.99	1.78%	18.67%
445.gobmk	43714	5	530.79	0.74	541.47	0.72	1162.91	0.63	2.01%	119.09%
456.hmmer	15563	2	350.59	1.31	357.65	0.17	446.69	1.78	2.01%	27.41%
458.sjeng	10502	1	629.40	3.04	638.24	1.02	1496.96	3.06	1.40%	137.84%
464.h264ref	24189	3	604.54	0.74	613.95	0.93	1008.73	3.57	1.56%	66.86%
471.omnetpp	47069	1	342.99	0.64	350.47	0.12	641.45	1.97	2.18%	87.01%
473.astar	6534	1	439.59	0.77	446.95	0.59	670.12	4.81	1.67%	52.44%

Table 1: Results of running coverage tools on the SPEC 2006 CPU Integer benchmarks. See text for details.

tude more than LCC's. LCC also has lower overhead for BizTalk and ClientSimulator. We conclude that the managed code support for LCC is efficient.

5.4 Windows Native Binaries

To evaluate the robustness of LCB, we applied LCC to all the native binaries from an internal release of Windows 8. We integrated and ran LCC on a subset of system tests in the standard Windows test environment. The goal of this experiment was to check if LCC can robustly handle a variety of executables, including kernel-mode drivers that are loaded during the operating system boot up. Another goal of this experiment was to ensure that LCC does not introduce test failures either due to implementation bugs or due to the timing perturbation introduced by the firing of breakpoints.

The system tests ran for a total of 4 hours on 17 machines. We repeated the test for different system builds: 32-bit and 64-bit x86 binaries, and ARM binaries. The size of the binaries covered ranges from 70 basic blocks to ~$1,000,000$. All tests completed successfully with no spurious failures or performance regressions.

To compare coverage, we ran the same tests with the `bbcover` tool. Figure 6 shows the difference in coverage achieved by the two tools. Of the 665 binaries, `bbcover` didn't produce coverage for 45 binaries because its overhead caused those tests to time out, thereby failing them. Therefore, the figure reports the coverage for the remaining 620 binaries. The binaries in the x-axis are ordered by the coverage achieved with `bbcover`.

As the tests are highly timing dependent and involve several boot-reboot cycles, there can be up to 20% difference in coverage across runs. Despite this nondeterminism, Figure 6 shows a clear trend. For all but 40 binaries, LCC reports more coverage than `bbcover`. This increased coverage is due to the fact that tests that time out or fail under `bbcover`, due to problems in relocation or excessive runtime, run to completion with LCC. For a small number of cases, LCC reports less coverage than `bbcover` due to test non-determinism.

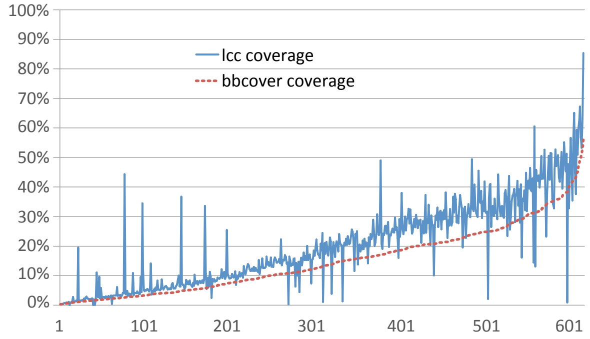


Figure 6: Difference between the coverage reported by LCC vs. `bbcover` (y-axis) for 620 Windows binaries (x-axis).

6 Cold Block Tracing

In this section, we extend LCC to create a simple tracing/logging facility for cold basic blocks, using two different strategies. First, we store the order in which breakpoints fire in a log. This reflects a compressed form of an execution trace where all instances of a basic block beyond its first execution are dropped. The size of this log is bounded by the size of the program. We call this "single-shot logging", since a basic block identifier will appear at most once in the log. Second, we set the `R` parameter to infinity in the BfS algorithm (§3), to periodically refresh breakpoints on all basic blocks. With this option enabled, the size of the log file is proportional to the length of program execution rather than program size. Next, we discuss these two strategies in detail.

	base	*lcc*	*per0.5*		*per5.0*		*lcc*	*per0.5*		*per5.0*	
Test #	(sec.)	(sec.)	(sec.)	overhead	(sec.)	overhead	blocks	blocks	Growth	blocks	Growth
21	31.46	32.94	36.29	15.33%	33.10	5.20%	28731	85459	2.97	46356	1.61
52	31.41	31.47	35.55	13.17%	32.72	4.15%	26506	72886	2.75	42359	1.60
12	46.95	47.38	54.39	15.84%	49.02	4.40%	25472	113730	4.46	45437	1.78
57	48.07	48.14	54.76	13.91%	49.69	3.37%	25525	114658	4.49	45882	1.80
43	57.22	56.89	65.44	14.36%	60.13	5.07%	27264	129836	4.76	51316	1.88
65	61.53	61.19	70.74	14.96%	63.90	3.84%	25175	138819	5.51	48581	1.93
14	70.74	71.63	81.27	14.88%	74.22	4.91%	25690	149329	5.81	59058	2.30
62	89.14	89.31	101.96	14.38%	94.07	5.54%	28191	185079	6.57	67408	2.39
29	155.09	156.63	175.82	13.37%	164.88	6.31%	25522	269179	10.55	72864	2.85

Table 2: Periodic logging of Z3 on tests that execute 30 seconds or more.

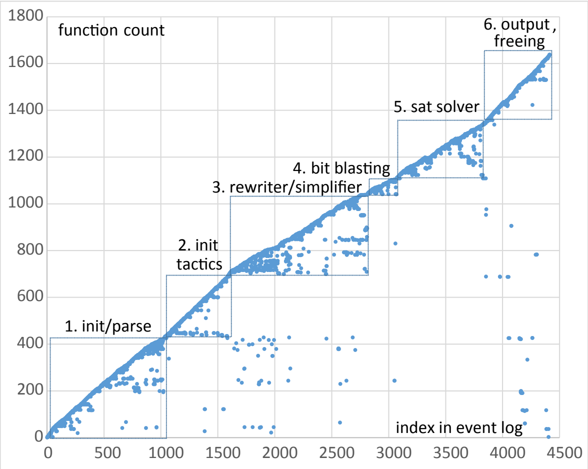


Figure 7: Scatterplot of the coverage log for Z3 test 15 with single-shot code coverage. The base execution of Z3 on test 15 took 20 seconds.

6.1 Single-Shot Code Coverage Logs

The additional cost to log basic block identifiers to a file is negligible for long executions and can be ameliorated by writing the log to a memory buffer, which in the case of single-shot logging is bounded by the size of the program. We give details on the overhead of logging when we consider periodic logging.

We can view a code coverage log as a sequence of events $(i, b(i))$, where i is the index of the event in the log and $b(i)$ is the block id. Such information about the relative ordering of the first execution of each basic block can be useful for identifying phases in a program's execution. Each basic block b has associated symbol information, including an identifier of the function $f(b)$ in which it resides. We assign to each function f a count $c(f)$ which corresponds to the number of unique functions that appear before it in the log.

Figure 7 shows, for a single execution of Z3, a scatter plot that contains one point for each log entry (executed basic block) with index i (x-axis), where the y-axis is the count of the function containing block, namely $c(f(b(i)))$. The scatterplot shows there are about 4500 events in the log. Phases are identified naturally by the pattern of "lower triangles" in which the blocks of a set of functions execute close together temporally. In the plot of the Z3 execution, we have highlighted six phases: (1) initialization of basic Z3 data structures and parsing of the input formula; (2) initialization of Z3's tactics that determine which decision procedures it will use to solve the given formula; (3) general rewriting and simplification; (4) bit blasting of bit-vector formula to propositional logic; (5) the SAT solver; (6) output of information and freeing of data structures.

This simple analysis shows that a one-shot log can be used to naturally identify sets of related functions since it provides an interprocedural view of control-flow behavior. We intend to use this information to identify program portions with performance bottlenecks and to improve job scheduling inside a datacenter [26, 11].

6.2 Periodic Logs

While one-shot code coverage logs are cheap to collect, there are many other (cold) traces the program will execute that will not be observed with the one-shot approach. To collect such information, we can periodically refresh the breakpoints on all basic blocks, as supported by the LCB framework.

Figure 8 shows the scatterplot of the execution log of Z3 run on the same test as in Figure 7, but with breakpoints refreshed every half second. From this plot, we can see that the SAT solver accounts for most of the log. Furthermore, notice that compared to the one-shot log in Figure 7, we see the interplay between the code of the SAT solver in phase 5 and the code of functions executed early on (during phase 1), which represent various commonly used data structures. We also observe more

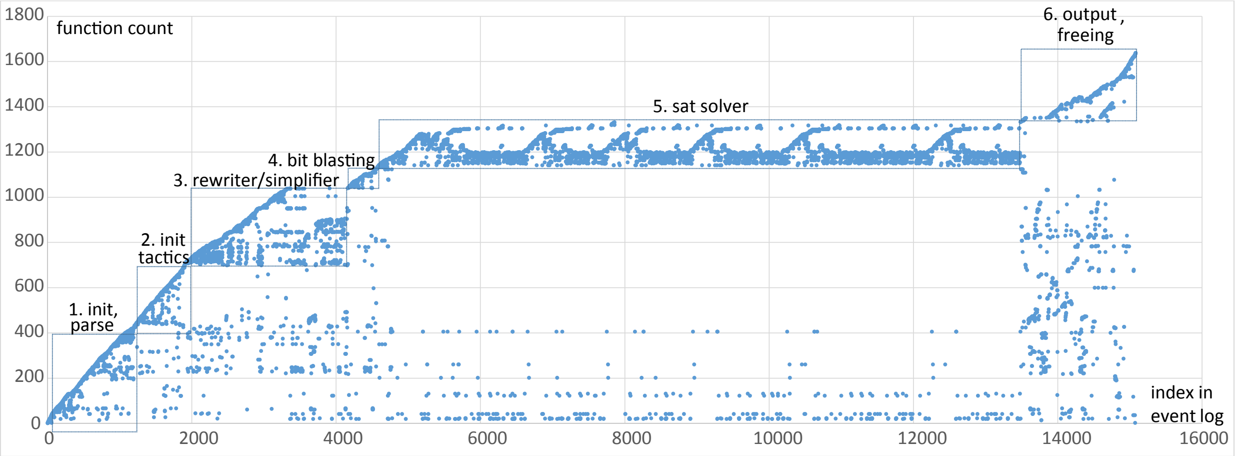


Figure 8: Scatterplot of the coverage log for Z3 test 15 with breakpoints are refreshed every .5 seconds.

activity between the functions in the final phase and the rest of the code (except for the SAT solver).

To evaluate the time and space costs of periodic logging, we selected all the 9 Z3 tests that execute 30 seconds or more in the base configuration. In our first experiment, we refresh all 439,927 breakpoints every half second (configuration *per0.5*), measure the overhead as well as the number of breakpoint firings. Note that the time to set all breakpoints is around 0.1 seconds, expected overheads range from 6 (for a 30 second test) to 30 seconds (for the longest test at around 150 seconds).

Table 2 shows the 9 tests ordered in increasing order of base execution time. As expected, we see that the execution times for *per0.5* increase execution overhead by 4 seconds on the low end (test 21) compared to *base* and 20 seconds on the high end (test 29). While refreshing breakpoints twice a second significantly increases the overhead compared to the single-shot logging of the *lcc* configuration, it is still less expensive than the `bbcover` tool (which doesn't log). Not surprisingly, the size of the periodic log (column "*per0.5* blocks") compared to that of one-shot logging ("*lcc* blocks") is substantial (ranging from a growth of 2.97× to 10.55×).

In the second experiment, we refresh the breakpoints every 5 seconds (configuration *per5.0*), resulting in runtimes closer to that of *lcc* than *per0.5*, and reducing the growth rate of the periodic log substantially.

7 Related Work

Once debuggers gave programmers the ability to set and remove breakpoints on instructions [19], the idea of using a one-shot breakpoint to prove that an instruction was executed (or covered) by a test was born. The rest is just a "simple matter of coding". The first tool we found that uses one-shot breakpoints to collect code coverage is the Coverage Assistant in the IBM Application Testing Collection [4] which mentions "minimal" overheads but does not provide implementation specifics.

DataCollider [18] uses hardware breakpoints to sample memory accesses and detect data races, therefore, it uses a small number of breakpoints at a time (e.g. 4 in x86 processors). Conversely, bias-free sampling requires a large number of breakpoints—linear in the size of the program—to be handled efficiently, which LCB does.

Residual test coverage [25] places coverage probes for deployed software only on statements that have not been covered by pre-deployment testing, but these probes are not removed during execution when they have fired.

Tikir et al.'s work on efficient code coverage [29] uses the dynamic instrumentation capabilities of the DynInst system [7] to add and remove code coverage probes at run-time. While efficient, such approaches suffer from the problem that basic blocks that are smaller than the jump instruction (5 bytes on x86) cannot be monitored without sophisticated fixup of code that branches to the code following the basic block. In addition, special care has to be taken to safely perform the dynamic rewriting of the branch instruction in the presence of concurrent threads. For instance, DynInst temporarily blocks all the threads in the program before removing the instrumentation to ensure that all threads either see the instruction before or after the instrumentation.

The Pin framework [22] provides a virtual machine and trace-based JIT to dynamically rewrite code as it executes, with a code cache to avoid rewriting the same code multiple times. The overhead of Pin without any probes added is around 60% for integer benchmarks. The code cache already provides a form of code coverage as the presence of code in the cache means it has been executed.

Static instrumentation tools like PureCoverage [3], BullseyeCoverage [1], and Gcov [13] statically modify program source code to insert instrumentation that will be present in the program throughout its lifetime. These tools can also be used to determine infrequently executed code, albeit at the expense of always triggering the instrumentation for frequently-executed code.

THeME [30] is a coverage measurement tool that leverages hardware monitors and static analysis to measure coverage. THeME's average overhead is 5% (with a maximum overhead of up to 30%), however it can determine only up to 90% of the actual coverage. LCB has similar average overhead as THeME, but it fully accurately determines the actual coverage. Furthermore, LCB can be used to obtain multi-shot periodic logs.

Symbolic execution [20, 8] can be used to achieve high coverage in the face of cold code paths. In particular, symbolic execution can explore program paths that remain unexplored after regular testing, to increase coverage. However, symbolic execution is typically costly, and therefore, it is more suited to be used as an in-house testing method. Developers can employ symbolic execution in conjunction with BfS; the latter can be used in the field thanks to its low overhead.

8 Conclusion

Bias-free sampling of basic blocks provides a low overhead way to quickly identify and trace cold code at runtime. Its efficient implementation via breakpoints has numerous advantages over instrumentation-based approaches to monitoring. We demonstrated the application of bias-free sampling to code coverage and its extension to periodic logging, with reasonable overheads and little in the way of optimization.

Acknowledgments

We would like to thank our anonymous reviewers, and Wolfram Schulte, Chandra Prasad, Danny van Velzen, Edouard Bugnion, Jonas Wagner, and Silviu Andrica for their insightful feedback and generous help in building LCB and improving this paper. Baris Kasikci was supported in part by ERC Starting Grant No. 278656.

References

[1] BullseyeCoverage. http://www.bullseye.com/productInfo.html.

[2] Common Language Runtime. http://msdn.microsoft.com/en-us/library/8bs2ecf4%28v=vs.110%29.aspx.

[3] IBM Rational PureCoverage. ftp://ftp.software.ibm.com/software/rational/docs/v2003/purecov.

[4] Application testing collection for mvs/esa and os/390 user's guide. http://publibfp.dhe.ibm.com/cgi-bin/bookmgr/Shelves/atgsh001, January 1997.

[5] T. Ball. The concept of dynamic analysis. In *FSE*, 1999.

[6] S. Bhansali, W.-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE*, 2006.

[7] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *HPCA*, 14, 2000.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[9] F. Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[11] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. ASPLOS '13, 2013.

[12] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, 2002.

[13] GCC coverage testing tool, 2010. http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[14] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE: whitebox fuzzing for security testing. *Commun. ACM*, 55(3), 2012.

[15] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *OSDI*, 2008.

[16] M. Hauswirth and T. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, 2004.

[17] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *FFDO: Feedback-Directed and Dynamic Optimization*, December 2001.

[18] S. B. John Erickson, Madanlal Musuvathi and K. Olynyk. Effective data-race detection for the kernel. In *OSDI*, 2010.

[19] W. H. Josephs. An on-line machine language debugger for os/360. In *Proceedings of the Fall Joint Computer Conference (AFIPS'69)*, 1969.

[20] J. C. King. Symbolic execution and program testing. *Comm. ACM*, 1976.

[21] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. PIN: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[23] P. D. Marinescu and G. Candea. LFI: A practical and general library-level fault injector. In *DSN*, 2009.

[24] B. G. R. Matthew Arnold. A framework for reducing the cost of instrumented code. In *PLDI*. ACM, 2001.

[25] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *ICSE*, 1999.

[26] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. EuroSys '13, 2013.

[27] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[28] J. R. L. Thomas Ball. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4), July 1994.

[29] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA*, 2002.

[30] K. Walcott-Justice, J. Mars, and M. L. Soffa. Theme: A system for testing by hardware monitoring events. ISSTA 2012, 2012.

Gestalt: Fast, unified fault localization for networked systems

Radhika Niranjan Mysore[1], Ratul Mahajan[2], Amin Vahdat[1] and George Varghese[2]

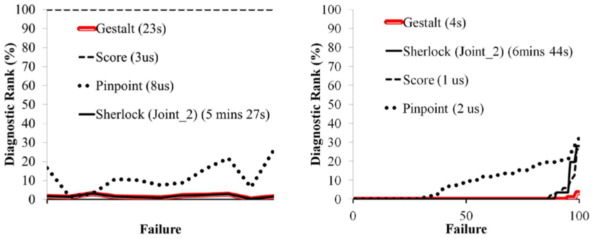
[1]Google [2]Microsoft Research

Abstract— We show that the performance of existing fault localization algorithms differs markedly for different networks; and no algorithm simultaneously provides high localization accuracy and low computational overhead. We develop a framework to explain these behaviors by anatomizing the algorithms with respect to six important characteristics of real networks, such as uncertain dependencies, noise, and covering relationships. We use this analysis to develop Gestalt, a new algorithm that combines the best elements of existing ones and includes a new technique to explore the space of fault hypotheses. We run experiments on three real, diverse networks. For each, Gestalt has either significantly higher localization accuracy or an order of magnitude lower running time. For example, when applied to the Lync messaging system that is used widely within corporations, Gestalt localizes faults with the same accuracy as Sherlock, while reducing fault localization time from days to 23 seconds.

1. Introduction

Consider a large system of components such as routers and servers interconnected by network paths. This system could be for audio, video, and text messaging (e.g., Lync [2]), for email (e.g., Microsoft Exchange), or even for simple packet delivery (e.g., Abilene). When transactions such as connection requests fail, a *fault-localization* tool helps identify likely faulty components. An effective tool allows operators to quickly replace faulty components or implement work-arounds, thus increasing the availability of mission-critical networked system.

As an example, we conducted a survey of call failures in the Lync messaging system deployed inside a large corporation. We found that the median time for diagnosis, which was largely manual, was around 8 hours because the operators had to carefully identify the faulty component from a large number of possibilities. This time-consuming process is frustrating and leads to significant productivity loss for other employees. A good fault localization tool that can identify a short list of potential suspects in a short amount of time would greatly reduce diagnosis time. Later in the paper, we will show how our tool, Gestalt, reduces by 60x the number of components that an operator must consider for diagnosis; and it has a median running time of under 30 seconds.



(a) Lync (real failures) (b) Exchange (simulated failures)

Figure 1: Applying different algorithms to two systems. Legend shows median time to completion.

Of course, we are not the first to realize the importance of fault localization, and other researchers have developed many algorithms (e.g., [3, 6, 8, 11, 13, 14, 16–18]).

However, we have consistently heard from operators (e.g., at Google and Microsoft) that the effectiveness of existing fault localization algorithms in terms of running times and accuracy depends on the network. There are no studies that connect network characteristics to the choice of algorithm, making it difficult to determine an appropriate algorithm for a given network. Figure 1 illustrates this difficulty by running three prior algorithms on two different networks. We picked these algorithms because they use disparate techniques. In the graphs, the *y*-axis is the diagnostic rank, which is the percentage of network components deemed more likely culprits than the components that actually failed; thus, lower values are better[1]. The failures are sorted by diagnostic rank. We

[1] In information retrieval terms, diagnostic rank includes the

provide more experimental details in §9,

The left graph shows the results for the Lync deployment mentioned above. We see that the algorithms perform differently. Sherlock [6](modified) does best, and SCORE [17] does worst. The right graph shows the results for simulated failures in an Exchange deployment [9]. We see that the algorithms exhibit different relative performance. SCORE matches Sherlock, and Pinpoint does worst. Further, the appropriate approach for the two networks differs—Sherlock for Lync, and SCORE for Exchange as it combines high localization accuracy and fast running time.

There is also a tradeoff between localization accuracy in the presence of impairments such as noise and computational cost for large networks. This tradeoff can be seen in Figure 1. While SCORE runs in a few microseconds, it localizes faults poorly for Lync. On the other hand, while Sherlock [6] has good performance for both networks, it takes a long time. In large networks, this time can be days. Running time matters because it directly influences time to recovery. For a large network like Lync, ideally we would want a localization algorithm with accuracy closer to Sherlock but runtime closer to SCORE.

Rather than simply developing yet another localization algorithm with its own tradeoffs, we first develop a framework to understand the design space and answer the basic question: When is a given fault localization approach better and why? We observe that existing fault localization algorithms can be anatomized into three parts that correspond to how they *i*) model the system, *ii*) compute the likelihood of a component failure, and *iii*) explore the state space of potential failures. Delineating the choices made by an algorithm for each part enables systematic analysis of the algorithm's behavior.

Our anatomization also explains phenomena found empirically, but not fully explained, in existing work. For example, Kompella et al. [18] discover that noise leads SCORE to produce many false positives; they then suggest mitigation through additional heuristics. By contrast, we show that certain design choices of SCORE are inherently sensitive to noise, and changing these would lead to more robust fault localization than the suggested heuristic. As a second example, Pinpoint was found to have poor accuracy for simultaneous failures [8]. We show that this problem is fundamentally caused by how Pinpoint explores the state space of failures.

We use our understanding to devise a new fault localization algorithm, called Gestalt. Gestalt combines the best features of existing algorithms to work well in many networks and conditions. While Gestalt benefits from reusing existing components, we also introduce a new method for exploring the space of potential failures. Our method navigates a continuum between the extremes of greedy failure hypothesis exploration (e.g., SCORE) and combinatorial exploration (e.g., Sherlock).

Experiments on three real, diverse networks show that Gestalt simultaneously provides high localization accuracy and low computational cost. For instance, in Figure 1, we can see that Gestalt has higher accuracy than SCORE and Pinpoint; its accuracy is similar to Sherlock, but its running time is an order of magnitude lower.

In summary, this paper contributes a new fault localization algorithm that simultaneously provides high localization accuracy and low running time for a range of networks. Its design is not driven by our intuition alone, but by anatomization of the design space of fault localization algorithms. and by analysis of the ability of the design choices of existing algorithms to handle various characteristics of real networks (e.g., noise).Our analysis framework also explains why certain algorithms work well for some networks and not for others.

2. Related Work

Network diagnosis can be thought of as having two phases. The first processes available information (e.g., log files, passive or active measurements) to estimate system operation and is often used to *detect* faults. Several system-specific techniques exist for this phase [5, 9–11, 15, 19–21, 23–25]. Its output is often fed to a second phase that *localizes* faults. Localization identifies which system components are likely to blame for failing transactions.

Fault localization techniques are extremely valuable because information on component health may not be easily available in large networks and manual localization can lead to several hours of downtime. Even where component health information is available, it may be incorrect (as in the case of "gray failures" in which a failed component appears functional to liveness probes) or insufficient towards identifying culprits for failing transactions [6]. Fault localization has also been studied widely [3,6,8,11,13,14,16–18,26,27]. We focus on this second phase and ask: *given information from the first phase, which fault localization algorithm gives the best accuracy with the lowest overhead, and why?*.

Some diagnostic tools like [21, 23, 24] leave fault localization to a knowledgeable network operator and aim to provide the operator with a reduced dependency graph for a particular failure. While this is different from what we call fault localization in this paper, the automated fault-localization techniques we discuss can be used in those tools as well, to narrow down the list of suspects.

Steinder and Sethi [28] suvey the fault localization landscape but consider each approach separately. To the

impact of both precision and recall. It will be high if components deemed more likely are not actual failures (poor precision) or if actual failures are deemed unlikely (poor recall).

best of our knowledge, ours is the first work to analyze the design space for fault localization, and to use this insight to propose a better fault localization tool Gestalt.

3. Fault Localization Anatomy

We consider the following common fault localization scenario. The network is composed of components such as routers and servers. The success of a transaction in the network depends on the health of the components it uses. The goal of fault localization is to identify components that are likely responsible for failing transactions. While we use the term transaction for simplicity in this paper, it can be any indicator of network health (e.g., link load) for which we want to find the culprit component.

More formally, the state of the network is represented by a vector I with one element $I[j]$ per network component that represents the health of component j. Let O be a vector of observation data such that $O[k]$ represents whether transaction k succeeded. For example, O could represent the results of pings between different sources and destinations. The broad goal is to infer likely values of I that explain the observations O. Specifically, the fault localization algorithm outputs a sequence of possible state vectors $I_1, I_2, ..$ ordered in terms of likelihood.

We measure the goodness of an algorithm by its *diagnostic rank*: given ground truth about the components that failed denoted by I_{true}, the diagnostic rank is j if $I_{true} = I_j$ for some j in the output sequence; and n, the total number of possible state vectors, otherwise. For example, a network with two routers R and S and one link E between them will have a 3 element state vector denoting the states of R, S, and E respectively. Let us say that only router R has failed so $I_{true} = (F,U,U)$ where F denotes failed and U denotes up. If the output of the fault localization algorithm is $(U,F,U),(F,U,U),(U,U,F)$ then the diagnostic rank on this instance of running fault localization is 2 because one other component failure (router S) has been considered more likely. Lower diagnostic rank implies fewer "false leads" that an operator must investigate. A second metric for an algorithm is the computation time required to produce the ranked list given the observation vector O.

We find that practical fault localization algorithms can be anatomized into three parts: a system model, a scoring function, and a state-space explorer. First, any fault localization algorithm needs information such as which components are used by each transaction, and possible failure correlations between component failures (e.g., a group of links in a load-balancing relationship). Thus, localization algorithms start with a **system model** S that predicts the observations produced when the system is in state I. System models in past work are often cast in the form of a *dependency graph* between transactions and components but there is considerable variety in the types of dependency graphs used (§4.1).

Second, in theory fault localization can be cast as a Bayesian inference problem. Given observation O, rank system states I based on $P_S(I|O)$, the probability that I led to O when passed through the system model S. However, even approximate Bayesian inference [12, 22] can seldom handle the complexity of large networks [13]. So practical algorithms use a heuristic scoring function *Score* that maps each component to a metric that represents the likelihood of that component failing. The underlying assumption is that for two system states I_i and I_j and respective observations O_i and O_j predicted by S: $P_S(I_i|O) \geq P_S(I_j|O)$ when $Score(O_i,O) \geq Score(O_j,O)$, where O is the actual observation vector. This **scoring function** is the second part of the pattern.

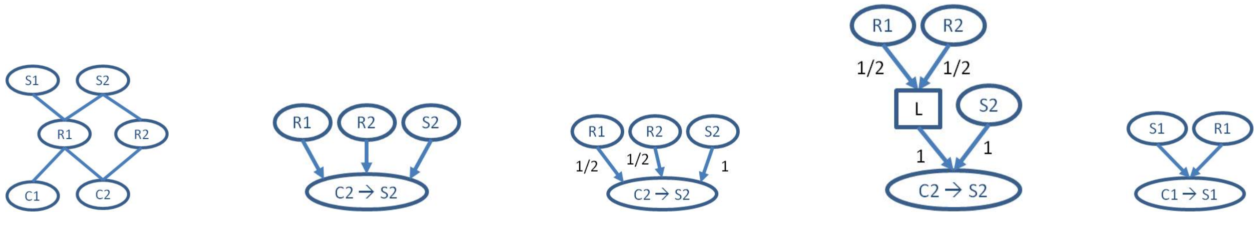
Finally, given the system model and scoring function the final job of a fault localization algorithm is to list and evaluate states that are more likely to produce the given observation vector. But system states can be exponential in the number of components since any combination of components can fail. Thus, localization algorithms have a third part that we call **state space exploration** in which heuristic algorithms are used to explore system states, balancing computation time with accuracy.

We do not claim that this pattern fits all possible fault localization algorithms. It does not fit algorithms based on belief propagation [26, 27]; such algorithms are computationally expensive and have not been shown to work with real systems. However, as Table 3 shows, this pattern does capture algorithms that have been evaluated for real networks, despite considerable diversity in this set.

4. Design Choices for Localization

We map existing algorithms into the three-part pattern by describing the choices they make for each part. §4.1-4.3 describes the choices, and §4.4 provides the mapping.

Prior algorithms also use different representations such as binary [8, 17, 18] or probabilities [14]) for transaction and component states. We use the 3-value representation from Sherlock [6] as it can model all prior representations. Specifically, the state of a component or transaction is a 3-tuple, $(p^{up}, p^{troubled}, p^{down})$, where p^{up} is the probability of being healthy, p^{down} that of having failed, and $p^{troubled}$ that of experiencing partial failure; p^{up}+$p^{troubled}$+p^{down}=1. The state of a completely successful or failed transaction or component is (1,0,0) or (0,0,1); other tuples represent intermediate degrees of health. A monitoring engine determines the state of a transaction in a system specific way; for example, a transaction that completes but takes a long time may be assigned $p^{troubled} > 0$.



(a) Network (b) DTL: $C_2 \to S_2$ (c) PTL: $C_2 \to S_2$ (d) PML: $C_2 \to S_2$ (e) $C_1 \to S_1$

Figure 2: An example network and models for two transactions.

	Failed component (s)			
	R1	R2	S2	R1&R2
DTL	0	0	0	0
PTL	1/2	1/2	0	3/4
PML	1/2	1/2	0	0

Table 1: Transaction state (p^{up}) predicted by different models for transaction $C_2 \to S_2$ in Figure 2

4.1 System Model

A system model represents the impact of network components on transactions. It can be encoded as a directed graph, where an edge from A to B implies that A impacts B. Three types of system models have been used by prior localization algorithms:

1. Deterministic Two Level (DTL) is a two-level model in which the top level corresponds to system components and the bottom level to transactions. Components connect to transactions they impact. The model assumes that components independently impact dependent transactions, and a transaction fails if any of its parent components fails.

2. Probabilistic Two Level (PTL) is similar to DTL except that the impact is modeled as probabilistic. Component failure leads to transaction failure with some probability.

3. Probabilistic Multi Level (PML) can have more than two levels; intermediate levels help encode more complex relationships between components and transactions such as load balancing and failover.

The network in Figure 2(a) helps illustrate the three models. The network has two clients (C_1, C_2), two servers (S_1, S_2), two routers (R_1, R_2), and several links. Transactions are requests from a client to a server ($C_i \to S_j$). Each request uses the shortest path, based on hop count, between the client and server. Where multiple shortest paths are present, as for $C_2 \to S_2$, requests are load balanced across those paths.

Assume that the components of interest for diagnosis are the two routers and the two servers. Then, Figures 2(b)-(d) show the models for the transaction $C_2 \to S_2$. Different models predict different relationships between the failures of components and that of the transaction. These predictions are shown in Table 1. For simplicity, the table shows the value of p^{up}; $p^{down} = 1 - p^{up}$ and $p^{troubled} = 0$ in this example. DTL predicts that the transaction fails when any of the components upon which it relies fails. Thus, the transaction is (incorrectly) predicted as always failing even when only one of the routers fails. PTL provides a better approximation in that the transaction is not deemed to completely fail when only one of the router fails. However, it still does not correctly model the impact of both routers failing simultaneously. PML is able to correctly encode complex relationships. While this example shows how PML correctly captures load balancing, it can also model other relationships such as failover [6]. However, this higher modeling fidelity does not come for free; as we discuss later, PML models have higher computational overhead.

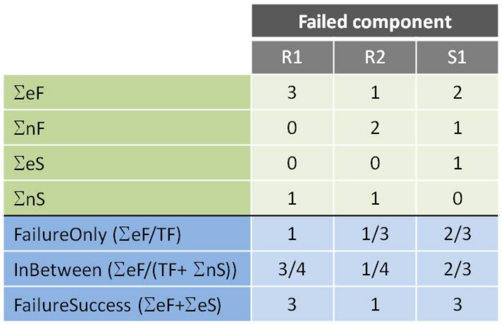
In this network, the three models for the other three types of transactions ($C_1 \to S_{\{1,2\}}, C_2 \to S_1$) are equivalent. The model for $C_1 \to S_1$ is shown in Figure 2(e)

4.2 Scoring function

Scoring functions evaluate how well the observation vector predicted by the system model for a system state matches the actual observation vector. Let $(p^{up}, p^{troubled}, p^{down})$ be the state of a transaction in the predicted observation vector. Let $(q^{up}, q^{troubled}, q^{down})$ be the actual state determined by the monitoring engine. Then, the computation of various scoring functions can be compactly explained using the following quantities:

Explained failure	$eF = p^{down} q^{down}$
Unexplained failure	$nF = (1 - p^{down}) q^{down}$
Explained success	$eS = p^{up} q^{up} + p^{troubled} q^{troubled}$
Unexplained success	$nS = (1 - p^{up}) q^{up} +$ $(1 - p^{troubled}) q^{troubled}$

eF is the extent to which the prediction explains the actual failure of the transaction, and nF measures the extent to which it does not. eS and nS have similar interpretations for successful transactions. We also define another quantity $TF = \Sigma(eF + nF)$, where the summation is over all elements of observation vectors. Because $eF + nF = q^{down}$, TF is the total number of failures in the actual observation vector.



	Failed component		
	R1	R2	S1
ΣeF	3	1	2
ΣnF	0	2	1
ΣeS	0	0	1
ΣnS	1	1	0
FailureOnly (ΣeF/TF)	1	1/3	2/3
InBetween (ΣeF/(TF+ ΣnS))	3/4	1/4	2/3
FailureSuccess (ΣeF+ΣeS)	3	1	3

Table 2: Score computed by different scoring functions for three possible failures.

Different scoring functions aggregate these basic quantities across observation elements in different ways. We find three classes of scoring functions:

1. FailureOnly (eF, TF): Such scoring functions only measure the extent to which a hypothesis explains actual failures. They thus use only eF and TF.

2. InBetween (eF, nS, TF): Such scoring functions only measure the extent to which a hypothesis explains failures and unexplained successes.

3. FailureSuccess (eF, eS): Such scoring functions measure *both* the extent to which a hypothesis explains failures and how well it explains successes.

Concrete instances of these classes are shown in Table 3. As expected, the score increases as eF and eS increase, and decreases when nF and nS increase. Given the large number of elements, each aggregates them in a way that is practical for high-dimensional spaces [4, 7].

Instead of analyzing every instance, in this paper we use a representative for each of the three classes. We find that the performance of different functions in a class is qualitatively similar. Our experiments use as representatives the functions used by SCORE (FailureOnly), Pinpoint (InBetween), and Sherlock (FailureSuccess).

To understand how different scoring functions can lead to different diagnoses, consider Figure 2 again. Assume that R_1 has failed and the actual state of four transactions is available to us. Two of these are $C_1 \rightarrow S_1$, both of which have failed (since they depend on $R1$); and the other two are $C_2 \rightarrow S_2$, one of which has failed (because it used $R1$, while the other used $R2$). Table 2 shows how the scoring functions evaluate three system states in which exactly one of R1, R2, and S1 has failed. The computation uses DTL for the system model. The top four rows show the values of the basic quantities. As an example, ΣeF is 3 in Column 1 because R1's failure correctly explains the three failed transactions; it is 1 in Column 2 because R2's failure explains the failure of only one transaction ($C_2 \rightarrow S_2$) and not of the two $C_1 \rightarrow S_1$ transactions.

The bottom three rows of the table show the scores of the three scoring functions for each failure. Even in this simple example, different scoring functions deem different failures as more or less likely. FailureOnly and InBetween deem $R1$ as the most likely failure that explains the observed data, FailureSuccess deems (incorrectly) that the data can be just as well be explained by the failure of $S1$. While it may appear that FailureSuccess is a poor choice, we show later that FailureSuccess actually works well in a variety of real networks.

4.3 State space exploration

State space exploration determines how the large space of possible system states (i.e., combinations of failed components) is explored. Prior work uses four types of explorers.

1. Independent explores only system states with exactly one component failure.

2. Joint$_k$ explores system states with at most k failures. It is a generalization of Independent (which is Joint$_1$).

3. Greedy set cover (Gsc) is an iterative method. In each iteration, a single component failure that explains the most failed transactions is chosen. Iterations repeat until all failed transactions are explained. Thus, it greedily computes the set of component failures that cover all failed transactions.

4. Hierarchical is also an iterative method. As in Gsc, in each iteration the component C that best explains the actual observations is chosen. However, a major difference is that if there are additional observations that C impacts, then these are added to the list of unexplained failures even if they were originally not marked as having failed in the input. Thus unlike Gsc, the set of unexplained failures need not decrease monotonically.

4.4 Mapping fault localization algorithms

Table 3 maps the fault localization portion of nine prior tools to our framework. Readers familiar with a tool may not immediately see how its computation maps to the choices shown because the original description uses different terminology. But in each case we have analytically and empirically verified the correctness of the mapping: composing the choices shown for the three parts leads to a matching computation (except for aspects mentioned below). Due to space constraints, we omit these verification experiments.

The last column lists aspects of the tool that are not captured in our framework. Most relate to pre- or post-processing data, e.g., candidate pre-selection removes irrelevant components at the start. The table does not list other suggestions by tool authors such as using priors that capture baseline component failure probabilities.

While the aspects we do not model are useful enhancements, they are complementary to the core localization algorithm. Our goal is to understand the behavior of fundamental choices made in the core algorithm. By

Tool	Target system	System Model	Scoring Function	State Space Exploration	Aspects not captured
Codebook [16]	Satellite comm. network	DTL,PTL	FailureSuccess ($\Sigma(eF+eS)$)	Independent	Codebook selection
MaxCoverage [18]	ISP backbone	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Candidate post-selection, Hypothesis selection
NetDiagnoser [11]	Intra-AS, multi-AS internetwork	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Candidate pre-selection
NetMedic [14]	Small enterprise network	PTL	FailureOnly (ΣeF)	Independent	Re-ranking
Pinpoint [8]	Internet services	DTL	InBetween ($\frac{\Sigma eF}{TF+\Sigma nS}$)	Hierarchical	
SCORE [17]	ISP backbone	DTL	FailureOnly($\frac{\Sigma eF}{TF}$)	Gsc	Threshold based hypothesis selection
Sherlock [6]	Large enterprise network	PML	FailureSuccess ($\prod(eF+eS)$)	Joint$_3$	Statistical significance test
Shrink [13]	IP network	PTL	FailureSuccess ($\prod(eF+eS)$)	Joint$_3$	
WebProfiler [3]	Web applications	DTL	InBetween ($\frac{\Sigma eF}{\Sigma nS+\Sigma eF}$)	Joint$_2$	Re-ranking

Table 3: Different fault localization algorithms mapped to our framework.

employing these choices, tools inherit their implications (§7) even when they use additional enhancements. Our paper abuses notation for simplicity; when we refer to a particular tool by name, we are referring to the computation that results from combining its three-part choices.

5. Network Characteristics

Localization algorithms must handle network characteristics that confound inference. We selected six such characteristics by simply asking: "what could go wrong with inference?" Clearly, dependency graph information can be incorrect (which we call uncertainty) and measurements may be wrongly recorded (which we call noise). We, and other researchers, have seen each characteristic empirically: e.g., noise in Lync and uncertainty in Exchange. We make no claim that our six characteristics are exhaustive but only that they helped explain why inference in the real networks we studied was hard.

In detail, the six characteristics we study are:

1. Uncertainty Most networks have significant nondeterminism that makes the impact of a component failure on a transaction uncertain. For example, if a DNS translation is cached, a ping need not consult the DNS server; thus the DNS server failure does not impact the ping transaction if the entry is cached, but otherwise it does. This creates an uncertain dependency because the localization algorithm is not privy to DNS cache state. Load balancing is another common source of non-determinism e.g., $C_2 \rightarrow S_2$ transaction in Figure 2.

More precisely, if a component potentially (but not always) impacts a transaction failure, we call the dependency *uncertain*. A network whose system model contains uncertain edges is said to exhibit uncertainty. The degree of uncertainty is measured by the number of uncertain dependencies and the uncertainty of each dependency. Probabilistic models like PTL and PML can encode uncertainty while deterministic models cannot.

2. Observation noise So far, we assumed that observations are measured correctly. However, in practice, pings could be received correctly but lost during transmission to the stored log: thus an "up" transaction can be incorrectly marked as "down". Errors can also occur in reverse. In Lync, for example, the monitoring system measures properties of received voice call data to determine that a voice call is working; however, the voice call may still have been unacceptable to the humans involved. Both problems have been encountered in real networks [3, 11, 17, 18]. They can be viewed as introducing noise in the observation data that can lead sensitive localization algorithms astray. A network with 10% noise can be thought of as flipping 10% of the transaction states before presentation to the localization algorithm.

3. Covering relationships In some systems, when a particular component is used by a transaction, other components are used as well. For example, when a link participates in an end-to-end path, so do the two routers on either end. More precisely, component C covers component D if the set of transactions that C impacts is a superset of the transactions that D impacts.

Covering relationships confuse fault localization because any failed transaction explained by the covered component (link) can also be explained by the covering component (router). Other observations can be used to differentiate such failures; when a router fails, there may be path failures that do not involve the covered link. But some fault localization methods are better than others at making this distinction.

4. Simultaneous failures Diagnosing multiple, simultaneous failures is a well-known hurdle. Investigating k simultaneous failures among n components potentially requires examining $O(n^k)$ combinations of components. For example, in Lync, even if we limit localization to components that are actively involved in current transactions, the number of components can be around 600; naively considering 3 simultaneous failures as in *Joint*$_3$ can take days to run. The key characteristic is the maximum number s of simultaneous failures; the operator must feel that more than s simultaneous failures are extremely unlikely in practice.

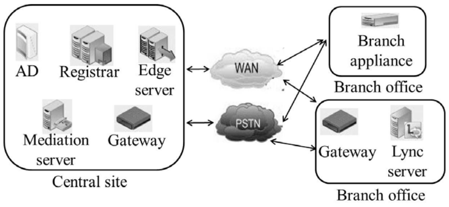


Figure 3: Lync architecture.

5. Collective impact So far, we assumed that a *single* component failure affects a transaction in possibly uncertain fashion. However, many networks exhibit a more complex dependency between a transaction and a *set* of components; the transaction's success depends on the collective health of the components in the set. For instance, when two servers are in a failover configuration, the transaction fails only if they both fail; otherwise, it succeeds. Collective impact is not limited to failover and load-balancing servers. Routers or links on the primary and backup paths in an IP network also have collective impact on message delivery. Multi-level models (e.g., PML) can model collective impact using additional logical nodes, but two-level models do not.[2]

6. Scale Network size impacts the speed of fault localization, which is key to fast recovery and high availability. Scale can be captured using the total number of components in the network and/or the typical number of observations fed to the localization algorithm. For Lync, the two numbers are 8000 and 2500.

6. Analysis methodology

Our goal is to analyze the relative merits of the choices made by various localization algorithms in the face of the network characteristics above. We do this by combining first principles reasoning and simulations of three diverse, real networks. This section describes our simulation method and the networks we study, and the next section presents our findings.

6.1 Simulation harness

In each simulation, we first select which system components fail. We then generate enough transactions—some of which fail due to the failed components—such that diagnosis is not limited by a lack of observations, as is true of large, busy networks [18, 21]. Finally, we feed these observations to the fault localization algorithm and obtain its output as a ranked list of likely failures.

Unless otherwise specified, the components to fail and the transaction endpoints are selected randomly. In practice, failures may not be random; we have verified that results are qualitatively similar for skewed failure distributions. In §9, we show that our findings agree with diagnosing real failures in Lync.

As is common, we quantify localization performance using *diagnostic rank* and *computation time*. Since diagnostic rank is the rank of components that have actually failed, it reflects the overhead of identifying real failures, assuming that operators investigate component failures in the order listed by the localization algorithm. Our simulation harness takes as input any network, any failure model, and any combination of localization methods.

6.2 Networks considered

To ensure that our findings are general, we study three real networks that are highly diverse in terms of their size, services offered, and network characteristics. The first network, Lync, supports interactive, peer-to-peer communication between users; the second, Exchange, uses a client-server communication model; and the third, Abilene, is an IP-based backbone. Each network has one or more challenging characteristics. For instance, Lync has significant noise and simultaneous failures while Exchange has significant uncertainty. To our knowledge we are the first to consider diagnosis in a Lync-like network.

1. Lync Lync is an enterprise communication system. that supports several communication modes, including instant messages, voice, video and conferencing. We focus on the peer-to-peer communication aspects of Lync. The main components of a Lync network are shown in Figure 3. Internal users are registered with registrars and authenticated with AD (active directory). Audio calls connect via mediation servers, and out of the enterprise into a PSTN (public switched telephone network) using gateway. Edge servers handle external calls. Branch offices connect to the main sites by a WAN and PSTN.

The deployment of Lync that we study spans many offices worldwide of a large enterprise. It has over 8K components and serves 22K users. We have information on the network topology and locations of users. For a two-month period, we also have information on failures from the network's trouble ticket database and on transactions from its monitoring engine.

2. Exchange Exchange is a popular email system. Its transactions include sending and receiving email and are based on client-server communication. Important components of an Exchange network include mail servers, DNS, and Active Directory(AD) servers.

We study the Exchange deployment used in [9], with 530 users across 5 regions. The network has 118 components. The number of hubs, mailboxes, DNS and AD servers in a region are proportional to the number of

[2] Our notion of collective impact differs from so called "correlated failures" in the literature which refers to components likely to fail together such as two servers are connected to the same power source.

users. AD servers are in a load balancing cluster; hubs, DNS and mailbox servers are in a failover configuration.

3. Abilene Abilene is an IP-based backbone that connects many academic institutions in the USA. The topology [1] that we use has 12 routers and 15 links, for a total of 27 network components. The workload used for Abilene consists of paths between randomly selected ingress and egress routers selected.

7. Analysis results

Table 4 summarizes our findings by qualitatively rating models, scoring functions and explorers on how they handle the six network characteristics. For each network characteristic (columns), it rates each method as good, OK, or poor. An empty (shaded) subcolumn for a characteristic implies that each row is qualitatively equivalent with respect to that characteristic. For instance, the choice of state space explorer has little impact on the ability to handle uncertainty. We focus on parts of the table where different options behave differently. Each such part highlights the relative merits of choices, and we use it later to guide the design of Gestalt.

7.1 Uncertainty

Uncertainty arises when the impact of a component on a transaction is not certain. Conventional wisdom is that deterministic models cannot handle uncertainty [6, 13, 14]. But we find that:

Finding 1 *In the presence of uncertainty, DTL suffices if the scoring function is FailureOnly.* Consider a DNS server *D* whose impact on a specific transaction, say ping 1, is uncertain. In DTL, this uncertainty must be resolved (since the model is binary) in favor of assuming impact; for instance, we must assume that ping 1 depends on the *D* even if it used a locally cached entry. (If we err in the opposite direction and assume that ping 1 does not depend on the *D*, we would never be able to implicate *D* if the cache is empty and *D* actually fails.)

If this assumption happens to be true, no harm is done. But if false (i.e., the transaction does not depend on the component), there are two concerns. First, consider the case when the the real failure was a different component; for example, ping 1 failed because some router *R* in the path failed and not because *D* failed. In that case, *D* may be considered a more likely cause of the failure of ping 1 than *R*; but this can increase the diagnostic rank of *R* by at most 1, which is insignificant.

The second, more important, concern is that the ability to diagnose the failure of the falsely connected component itself may be significantly diminished. For example, when *D* fails, other transactions, say ping 2 and ping 3, may succeed because they use cached entries. This can confuse the fault localization algorithm because it increases the number of unexplained successes *nS* attributed to *D*, and decreases *eS*, potentially increasing significantly the diagnostic rank of *D*.

FailureOnly functions are not hindered by the false connection because they use only *eF* and *nF* in computing their score. But FailureSuccess and InBetween are negatively impacted because they do use *eS* and *nS*.

Figure 4 provides empirical confirmation for this finding using Exchange which has significant uncertainty because of the use of DNS servers whose results can be cached. It plots the diagnostic rank for 1000 trials; in each trial, a single random failure is injected. Observe that DTL with FailureOnly handles uncertainty just as well as PML and PTL. By contrast, DTL with FailureSuccess has much worse diagnostic rank (50 versus 5 in some trials). An implication of Finding 1 is that if the network has only uncertainty, it can be best handled (with small computation time and comparable diagnostic rank) using DTL and FailureOnly.

7.2 Observation noise

Finding 2 *FailureSuccess is most robust to observation noise, followed by InBetween, and then by FailureOnly.* Intuitively, using more evidence and all available elements reduces sensitivity to noise. Noise turns successful transactions into apparent failures or vice versa. FailureOnly is the most impacted because it uses only failure elements. FailureSuccess is the least impacted as legitimate failures appearing as successes add to *eS* the same amount as that subtracted from *eF*, and vice versa.

Figure 6(a) confirms this behavior. We inject single failures in Abilene and introduce 0-50% noise. We run 100 trials for each noise level and plot the median diagnostic rank for each level. This graph uses DTL and Independent as the system model and state space explorer; the relative trends are similar with other combinations.

Finding 3 *Iterative state space explorers, Gsc and Hierarchical, are highly sensitive to noise.* This is because an erroneous inference (due to noise) made in an early iteration can cause future inferences to falter.

Figure 6(b) confirms this behavior. In this experiment, we introduced two independent failures in Abilene and 0-50% observation noise. The experiment uses DTL and FailureSuccess while varying the state space explorer; other combinations of model and scoring function produce similar trends. Figure 6(b) plots the median diagnostic rank across 100 trials. We see that Gsc and Hierarchical deteriorate with small amounts of noise. Finding 3 helps explain the extreme sensitivity of SCORE, which uses FailureOnly and Gsc, to noise, that prior work [18] empirically observed but did not fully explain.

7.3 Covering relationships

Recall that a component *C* covers a component *D* if

	Uncertainty	Observation Noise	Covering relationship	Simultaneous failures	Collective Impact	Scale
DTL	Good w/ FailureOnly. Poor w/ other scoring funcs.				Poor	Good
PTL	Good				Poor	OK
PML	Good				Good w/ Joint$_k$. Poor otherwise.	OK
FailureOnly(FO)	Good	Poor	Poor			Good
InBetween	Good w/ PTL, PML Poor with DTL	OK	Good			OK
FailureSuccess(FS)	Good w/ PTL, PML. Poor with DTL	Good	Good			OK
Independent(Ind)		Good		Poor	Poor	Good
Joint$_k$(Jt_k)		Good		Good (s≤k). Poor (s>k)	Good (c≤k). Poor (c>k)	Poor
Gsc		Poor		Good*	Poor	Good
Hierarchical		Poor		Poor	Poor	OK

Table 4: Effectiveness of diagnostic methods with respect to factors of interest. * depends on the network.

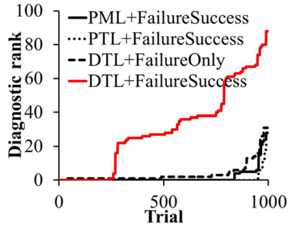


Figure 4: DTL can handle uncertainty when used with FailureOnly. [Exchange]

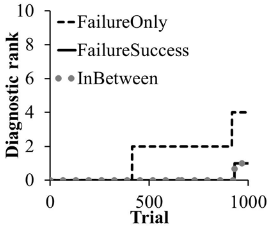
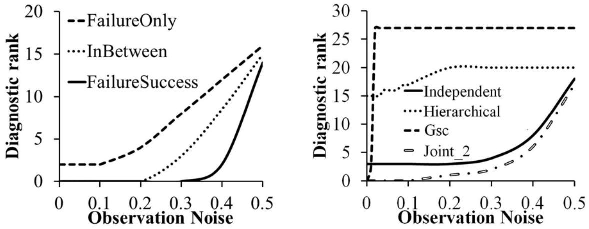


Figure 5: FailureOnly performs poorly for covering relationships. [Abilene]



(a) Scoring functions

(b) State space explorers

Figure 6: Sensitivity to observation noise. [Abilene]

the set of transactions that D impacts is a subset of those that C impacts. In other words, when a transaction that D impacts fails, it is impossible to distinguish a failure of C from that of D by looking only at failures.

Finding 4 *For covering relationships, FailureOnly scoring functions should not be used.* Other scoring functions (FailureSuccess, InBetween) can better disambiguate the failures of the covering and covered component because they use successful transactions (eS, nS) as well, and not only failed ones. For instance, consider a failed link. All failed transactions due to the link can also be explained by the failure of the attached routers. By using successful transactions that include the routers but not the failed link, the scoring function can assign a higher likelihood to link failure than router failure.

Figure 5 verifies Finding 4 by showing the results of an experiment using Abilene, which has many covering relationships. We randomly introduced a single failure in the network and diagnosed it using different scoring functions (combined with DTL and Independent). We see that FailureOnly has the worst performance with non-zero diagnostic rank in 60% of the trials while the other two methods have rank 0 most of the time.

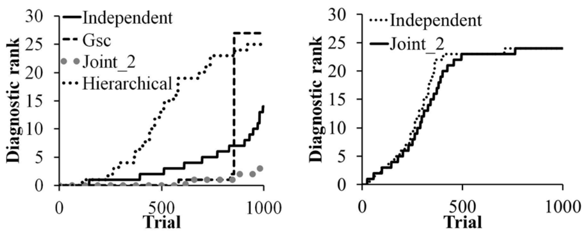
We note that FailureOnly has been used by several tools to diagnose ISP backbones [11, 17, 18], which have many covering relationships. Finding 4 suggests that the localization accuracy of these tools can be improved by changing their scoring function.

7.4 Simultaneous failures

We now discuss simultaneous failures of components that have *independent* impact on transactions. The next section discusses *collective* impact.

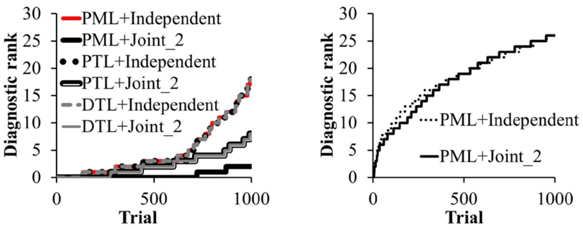
Finding 5 *For a small number of simultaneous failures (s≤k), Joint$_k$ is best and Hierarchical is worst.* The effectiveness of Joint$_k$ follows because it examines all system states with k or fewer failures. Hierarchical does poorly because its clustering approach forces it to explain more failures than needed. Suppose transactions O_1, O_2, O_3 have failed and component C explains O_1 and O_2 and no other component explains more failures. Suppose, however, that C also impacts transaction O_4. Then Hierarchical will add C to the cluster but will also add transaction O_4 as a new failed transaction to be explained by subsequent iterations. Intuitively, the onus of explaining more failures than those observed can lead Hierarchical astray.

Figure 7(a) shows the performance of different state space explorers when diagnosing two (randomly picked) simultaneous failures in Abilene. The graph uses PML



(a) 2 simultaneous failures (b) 3-4 simultaneous failures

Figure 7: Ability of state space explorers to handle simultaneous failures. [Abilene]



(a) 2 collective failures (b) 3 collective failures

Figure 8: Ability of a model+state space explorer to handle collective impact failures. [Abilene]

and FailureSuccess; other combinations produce similar trends. We see that Joint$_k$ is highly effective (rank 2 or less), and Hierarchical is poor (rank > 20 in 25% of trials). Gsc has bimodal behavior with a rank > 25 in a small fraction of trials. Closer investigation confirms that these trials involve the simultaneous failures of two components that together cover a third component.

Finding 5 explains why Pinpoint [8], which uses Hierarchical, has poor performance (see Figure 4 in [8]) for even two simultaneous failures, despite the handling of simultaneous failures being an explicit goal of Pinpoint. It suggests that replacing Hierarchical state space exploration in Pinpoint (with, say, *Joint*$_2$) while keeping the same system model and scoring function would improve Pinpoint's diagnosis of simultaneous failures.

7.5 Collective impact

We now study simultaneous failures of components that have a collective impact on transactions by being, for instance, in a load balancing or failover relationship. We find that in such cases, the choice of system model and state space explorer should be jointly made. We explored two cases: when the number s of failed components in a collection is small ($s \leq$k), and when it is large ($s>k$).

Finding 6 *For diagnosing a small number of simultaneous failures in a collection ($s \leq k$), combining PML and Joint$_k$ is most effective; any other system model or state space explorer leads to poor diagnosis.* This is because, among existing models, only PML can encode collective impact relationships. Other models represent approximations that can be far from reality. However, picking the right model is not enough. The state explorer must also consider simultaneous failure of these components. Among existing state space explorers, only Joint$_k$ has this property. Independent does not consider simultaneous failures, and Gsc and Hierarchical assume that components have independent impact.

Figure 8(a) demonstrates this behavior. We modeled failures among components with collective impact in Abilene as follows. Each trial randomly selects a pair of nodes that has two vertex-disjoint paths between them. For messages between these nodes, the two paths can be considered to be in a failover relationship with collective impact. We then introduced a randomly selected failure along each path. Thus, all messages sent between the pair of nodes will now fail. For 1000 such trials, the graph plots the diagnostic ranks of several combinations of system model and state space explorer. It uses FailureSuccess for scoring function, but others yield similar results. We omit results for Gsc and Hierarchical; they had worse performance than Independent. As we can see, only PML+Joint$_2$ is effective.

This result implies that half-way measures are insufficient for diagnosing collective impact failures. We must both model relationships (PML) and explore joint failures (Joint$_k$). Localization suffers if either choice is wrong. For example, Shrink [13] uses PTL with Joint$_k$ even though it targets IP networks which may have potentially many failover paths. Finding 6 suggests that Shrink would do better to replace PTL with PML.

8. Gestalt

The insights from the analysis above led us to develop Gestalt. It combines ideas from existing algorithms and also includes a new state space exploration method.

For the system model, Gestalt uses a hybrid between DTL and PML that combines the simplicity of DTL (fixed number of levels, deterministic edges) with the expressiveness of PML (ability to capture complex component relationships). Our model has three levels, where the top level corresponds to system components that can fail independently and the bottom level to transactions. An intermediate level captures collective impact of system components. Instead of encoding probabilistic impact on the edges, the intermediate node encodes the *function* that captures the nature of the collective impact. The domain of this function is the combinations of states of the parent nodes, and the range is the impact of each combination on the transaction. Figure 9(a) shows how Gestalt models the example in Figure 2a. The intermediate node I encodes the collective impact of R_1 and R_2.

Algorithm 1: Pseudocode for Gestalt

```
$H_{all}$ = {};
For each $hitRatio$ in $1, 0.95, \cdots 0$ do
    $H_{curr}$ = (); //current hypothesis
    $O_{unexp} = O_{all}$; //unexplained observations
    $H_{all}$ += GenHyp($1, O_{unexp}, hitRatio, H_{curr}$);
Return $H_{all}$

GenHyp(i, $O_{unexp}, hitRatio, H_{curr}$)
$H_{return} = \{H_{curr}\}$;
$C_{new}$ = NewCandidates($hitRatio, O_{unexp}$);
For each c in $C_{new}$
    $hyp_{new} = (hyp$, c);
    If (i == k)
        $H_{return}$ += $hyp_{new}$;
    Else
        $O_{exp}$ = ExpObs($hyp_{new}, O_{unexp}$);
        $H_{return}$ += GenHyp($i+1, O_{unexp} - O_{exp}, hitRatio,$
$hyp_{new}$);
Return $H_{return}$

NewCandidates($hitRatio, O_{unexp}$)
$C_{new}$ = {};
For each c in CandidatePool
    If (HitRatio(c) $\geq hitRatio$)
        $C_{new}$ += c;
$score_{max}$ = MaxScore($C_{new}, O_{unexp}$);
$score_{noise} = Noise_{thresh} \times |O_{unexp}|$;
For each c in $C_{new}$
    If (Score(c) $< score_{max} - score_{noise}$)
        $C_{new}$ −= c;
Return $C_{new}$
```

The function represented by I is shown in the figure, which shows values only for p_{up} (p_{down}=1−p_{up}).

While for this example, PML too has only three levels, Figure 9(b) illustrates the difference between PML and Gestalt. Here, to reach S, C spreads packets across $R1$ and $R2$, and $R2$ spreads across $R3$ and $R4$. Figures 9(c) and 9(d) show PML and Gestalt models for this network.

Another difference between PML and our model is how we capture single components with uncertain impact on a transaction (e.g., a DNS server whose responses may be cached). Gestalt models these with 3 levels too. An intermediate node captures the uncertainty from the component's state to its impact on the transaction. It may deem, for instance, that the transaction will succeed with some probability even if the component fails.

As scoring function, we use FailureSuccess because of its robustness to noise and covering relationships (Findings 2 and 3). By explicitly modeling uncertainty (unlike DTL), the combination of our model and FailureSucess is robust to uncertainty as well (Finding 1).

For state space exploration, we develop a method that has the localization accuracy of Joint$_k$ and the low computational overhead of Gsc. It is based on the following observations. Gsc is susceptible to covering relationships because many failure combinations can explain the observations and Gsc explores only a subset, ignoring others (Finding 5). Gsc is susceptible to noise because noise can make it pick a poor candidate and rule out other possibilities (Finding 3). The diagnostic accuracy of Joint$_k$ for collective impact failures stems from the fact that it explores combinations of at most k failures; exploring a smaller number does not help (Finding 6). But because its exploration is fully combinatorial, it has a high computational overhead.

Our new exploration method is shown in Algorithm 1. It takes two parameters as input. The first is $Noise_{thresh}$, the percentage of observation noise expected in the network, which can be estimated from historical data. Given ground truth (post resolution) about a failure and the transaction logs, the percentage of transactions that cannot be explained by the ground truth reflects the level of observation noise. In Lync, we found this to be around 10%. The second parameter is k, the maximum number of simultaneous failures expected in the network. It can also be gleaned from historical failure data.

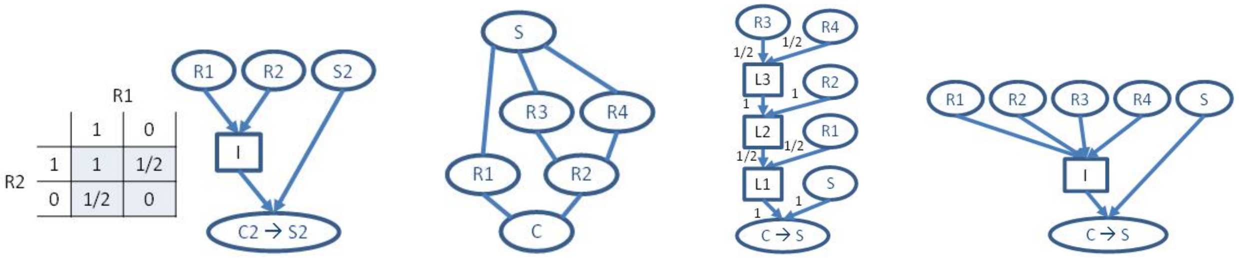
The candidate failures that we explore are single component failures and combinations of up to k components with collective impact. This candidate pool explicitly accounts for collective impact failures (making them diagnosable, unlike in Gsc). It is also much smaller than the pool considered by Joint$_k$ which includes all possible combinations of up to k failures. The output of the exploration is a ranked list of hypotheses, where each hypothesis is a set of at most k candidates from the pool.

These sets are computed separately for different thresholds of hit ratio [17]. The hit ratio of a candidate is the ratio of number of failed versus total transactions in which the component(s) participated. Iterating over candidates in decreasing order of hit ratios gives us a systematic way of exploring failures while focusing on more likely failures first because actual failures are likely to have larger hit ratios. Hit ratios are not used in the scoring function.

For a given hit ratio threshold, the hypothesis sets are built iteratively (i.e., not all possible sets are considered) in k steps. We start with the empty set. At each step, each set is forked into a number of child sets, where each child set has one additional candidate than the parent set.

The child candidates are computed as follows. Let O_{unexp} be the set of observations whose status cannot be explained by the parent set (i.e., the status does not match what would be predicted by the system model). Initially, when the parent set is empty, this set equals O_{all}, the set of all observations. Then, we first compute the score of each candidate in the entire pool with hit ratio higher than the current threshold. This computation uses the scoring function (FailureSuccess) and is done with respect to O_{unexp}. Candidates more likely to explain the as yet unexplained observations will have higher scores.

If there were no observation noise, candidates with the maximum score can be used as child candidates because they best explain the remaining unexplained ob-



(a) Gestalt model for Figure 2a

(b) Another example network

(c) PML for Fig. 9(b)

(d) Gestalt model for Figure 9(b)

Figure 9: Modeling in Gestalt

servations. But due to noisy observations, the score of actual failures may go down and the score of some other candidates may go up. By focusing only on candidates with the maximum score, we run the risk of excluding actual failures from the set, like Gsc.

We thus cast a wider net; the width of the net is proportional to expected noise. The score of the actual culprit can be expected to reduce due to observation noise by $score_{noise} = Noise_{thresh} \times |O_{unexp}|$. The selected candidates are those with scores higher than $score_{max} - score_{noise}$, where $score_{max}$ is the maximum score across all candidates. This reduces chances of missing actual failures. $Noise_{thresh}$ and k enable Gestalt to explore the continuum between Gsc and exhaustive search. $Noise_{thresh}$ set to 0 mimics Gsc (but handles covering relationship), and $Noise_{thresh}$ set to 100 mimics Joint$_k$.

9. Gestalt Evaluation

We now evaluate Gestalt and compare it to 3 existing algorithms that use very different techniques. We start with Lync and use the algorithms to diagnose real failures using real transactions available in system logs. Based on information from days prior to the failures we diagnose, we set $Noise_{thresh}$=10% and k=2 for Gestalt.

	Original recovery delay (days, hh:mm)	**# potential failed comps**	**Gestalt diagnostic rank**	**Gestalt run time (mm:ss)**
1	0,01:50	196	11	4:02
2	0,00:50	625	7	2:59
3	0,01:55	552	6	0:05
4	0,22:05	608	9	0:05
5	1,23:45	521	7	0:12
6	0,10:55	655	6	0:21
7	14,06:25	676	12	2:43
8	0,01:45	571	13	1:06
9	0,20:15	562	13	0:23
10	0,08:20	455	3	1:03

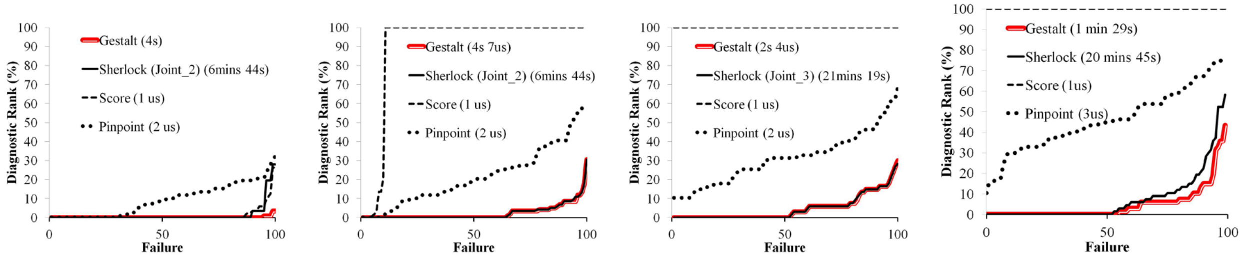
Table 5: Statistics for a sample of real failures in Lync.

Figure 1(a) shows the results for a number of failures seen in a two month period (the actual failure count is hidden for confidentiality). The legend shows the median running time for the algorithms on a 3 GHz dual-core PC. We see that SCORE and Pinpoint perform poorly. Gestalt and Sherlock perform similarly, but the running time of Gestalt is lower by more than an order of magnitude. This is despite the fact that we ran Sherlock with Joint$_2$. Using Joint$_3$, recommended in the original Sherlock paper [6], would have taken $\sim$ 20 hours per failure.

Table 5 provides more details for ten sample failures in the logs. We see that the time it took for the operators to manually diagnose these failures, reflected in the original recovery delay, was very high. The median time was around 8 hours, though it took more than a day for two failures. The primary reason for slow recovery time was the large diagnosis time due to the number of network components that had to be manually inspected[3]. The table lists the number of components involved in failing transactions as an estimate of the number of possible components that might need to be checked. Of course, using domain knowledge and expertise, an operator will only check a subset of these components; but the estimate underscores the challenge faced by operators today. We see that using Gestalt, the operator will have to check only 3-13 components before identifying the real culprits compared to 196-655 components for manual diagnosis, significantly reducing diagnosis time. The run time for Gestalt to whittle down the list of suspects by 1-2 orders of magnitude is at most a few minutes.

We next consider simulated failures in the Exchange network. Figures 10(a) and 10(b) show results for diagnosing one and two component simulated failures. We again used Joint$_2$ for Sherlock and k=2 and $Noise_{thresh}$=0 for Gestalt. As expected based on our earlier analysis, Score does very well for single failure scenarios, but suffers in two-failure scenarios due to covering relation-

[3] In Lync, once a problem was diagnosed, service was restored quickly by repair or diverting transactions around the failed component.



(a) Single failure (b) 2 simultaneous failures (c) 3 simultaneous failures (d) 4 simult. failures, 1% noise

Figure 10: Diagnostic efficacy of different algorithms with Exchange network with different number of failures.

ships. Sherlock and Gestalt do well for both cases, but Sherlock takes two orders of magnitude more time.

In order to experiment with more simultaneous failures and Joint$_3$, we reduced the size of the Exchange network by half (to 67 components). Figures 10(c) and 10(d) show the results for three failures and for four failures with 1% observation noise. In the latter case, we run Gestalt with $Noise_{thresh}$=1%. We see that Gestalt matches Sherlock's diagnostic accuracy for three failures, with running time that is two orders of magnitude faster. For four failures, Gestalt has better diagnostic accuracy than Sherlock because it accounts for noise. Its running time is still better by 20x, even though noise makes it explore more failure combinations.

We omit results for Abilene, but we found them to be qualitatively similar to those above. Gestalt had better diagnostic efficacy than SCORE and Pinpoint for all cases. Gestalt matched Sherlock's accuracy for most cases and exceeded it in the presence of noise and more than three simultaneous failures. Its running time was 1-2 orders of magnitude lower than Sherlock.

10. Conclusion

We presented Gestalt, a fault localization algorithm that borrows the best ideas from prior work and includes a new state space explorer that represents a continuum between greedy, low-accuracy exploration and combinatorial, high-overhead exploration. The result is an algorithm that simultaneously provides high localization accuracy and low overhead for a range of networks. Its design is guided by an analysis framework that anatomizes the design space of fault localization algorithms and explains how the design choices of existing algorithms interact with key characteristics of real networks. Beyond the specific algorithm it helped develop, we hope this framework takes a modest step towards understanding the *gestalt* of fault localization.

11. References

[1] Abilene Topology. `http://totem.run.montefiore.ulg.ac.be/files/examples/abilene/abilene.xml`, 2005.

[2] Microsoft Lync. http://en.wikipedia.org/wiki/Microsoft_Lync, 2012.

[3] AGARWAL, S., LIOGKAS, N., MOHAN, P., ET AL. Webprofiler: Cooperative diagnosis of web failures. In *COMSNET* (2010).

[4] AGGARWAL, C. C. Re-designing distance functions and distance-based applications for high dimensional data. In *SIGMOD Record* (2001).

[5] AGUILERA, M. K., MOGUL, J. C., WEINER, J. L., ET AL. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[6] BAHL, P., CHANDRA, R., GREENBERG, A., ET AL. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM* (2007).

[7] BEYER, K., GOLDSTEIN, J., RAMAKRISHNAN, R., ET AL. When is "nearest neighbor" meaningful? In *ICDT* (1999).

[8] CHEN, M., KICIMAN, E., FRATKIN, E., ET AL. Pinpoint: Problem determination in large, dynamic, internet services. In *IPDS* (2002).

[9] CHEN, X., ZHANG, M., MAO, M., ET AL. Automating network application dependency discovery:experiences, limitations, and new solutions. In *OSDI* (2008).

[10] CUNHA, T., TEIXEIRA, R., FEAMSTER, N., ET AL. Measurement methods for fast and accurate blackhole identification with binary tomography. In *IMC* (2009).

[11] DHAMDHEREY, A., TEIXEIRA, R., DOVROLIS, C., ET AL. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT* (2007).

[12] HECKERMAN, D. A tractable inference algorithm for diagnosing multiple diseases. In *Uncertainty in Artificial Intelligence* (1989).

[13] KANDULA, S., KATABI, D., AND VASSEURI, J.-P. Shrink: A tool for failure diagnosis in ip networks. In *MineNet workshop* (2005).

[14] KANDULA, S., MAHAJAN, R., VERKAIK, P., ET AL. Detailed diagnosis in computer networks. In *SIGCOMM* (2009).

[15] KATZ-BASSETT, E., MADHYASHTA, H. V., JOHN, J. P., ET AL. Studying black holes in the internet with hubble. In *NSDI* (2008).

[16] KLINGER, S., YEMINI, S., YEMINI, Y., ET AL. A coding approach to event correlation. In *International Symposium on Integrated Network Management* (1995).

[17] KOMPELLA, R. R., YATES, J., GREENBERG, A., ET AL. IP fault localization via risk modeling. In *NSDI* (2005).

[18] KOMPELLA, R. R., YATES, J., GREENBERG, A., ET AL. Detection and localization of network blackholes. In *Infocom* (2007).

[19] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing network-wide traffic anomalies. In *SIGCOMM* (2004).

[20] MAHAJAN, R., SPRING, N., WETHERALL, D., ET AL. User-level internet path diagnosis. In *SOSP* (2003).

[21] MAHIMKAR, A., GE, Z., SHAIKH, A., ET AL. Towards automated performance diagnosis in a large iptv network. In *SIGCOMM* (2009).

[22] MURPHY, K. P., WEISS, Y., AND JORDAN, M. I. Loopy belief-propagation for approximate inference: An empirical study. In *Uncertainty in Artificial Intelligence* (1999).

[23] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI* (2012).

[24] OLINER, A. J., AND AIKEN, A. Online detection of multi-component interactions in production systems. In *DSN* (2011).

[25] REYNOLDS, P., WEINER, J. L., MOGUL, J. C., ET AL. Performance debugging for distributed systems of black boxes. In *WWW* (2006).

[26] RISH, I. Distributed systems diagnosis using belief propagation. In *Allerton Conf. on Communication, Control and Computing* (2005).

[27] STEINDER, M., AND SETHI, A. Probabilistic fault localization in communication. In *IEEE/ACM Trans. Networking* (2004).

[28] STEINDER, M., AND SETHI, A. A survey of fault localization techniques in computer networks. In *Science of Computer Programming* (2004).

Insight: In-situ Online Service Failure Path Inference in Production Computing Infrastructures

Hiep Nguyen, Daniel J. Dean, Kamal Kc, Xiaohui Gu
Department of Computer Science
North Carolina State University
{hcnguye3,djdean2,kkc}@ncsu.edu, gu@csc.ncsu.edu

Abstract

Online service failures in production computing environments are notoriously difficult to debug. When those failures occur, the software developer often has little information for debugging. In this paper, we present Insight, a system that reproduces the execution path of a failed service request onsite immediately after a failure is detected. Upon a request failure is detected, Insight dynamically creates a shadow copy of the production server and performs *guided binary execution exploration* in the shadow node to gain useful knowledge on how the failure occurs. Insight leverages both environment data (e.g., input logs, configuration files, states of interacting components) and runtime outputs (e.g., console logs, system calls) to guide the failure path finding. Insight does not require source code access or any special system recording during normal production run. We have implemented Insight and evaluated it using 13 failures from a production cloud management system and 8 open source software systems. The experimental results show that Insight can successfully find high fidelity failure paths within a few minutes. Insight is light-weight and unobtrusive, making it practical for online service failure inference in the production computing environment.

1 Introduction

Although online services[1] are expected to be operational 24x7, recent production service outages [2, 1] show great challenge to meet such an expectation. Unfortunately, when those online services experience failures in a production computing environment, the software developer is often given little information for debugging.

Particularly, we focus on *non-crashing failures* where the server does not crash but fails to process some requests. Different from crash failures that often receive immediate attention, those non-crashing failures often go unnoticed. We observe that those failures are common in online services based on our experience with the virtual computing lab (VCL) [3] which is a production cloud computing infrastructure. Users who experience frequent service failures will be seriously discouraged to use the service again. Most production servers are well engineered to avoid fatal crash failures and strive to capture all the request failures with error messages. However, those error messages do not tell us *why* a service request has failed and can be misleading sometimes [7, 36].

To debug a production-run failure, software developers generally need to reproduce the failure at the developer-site to understand what happened during the production run in order to infer the root cause. Much effort has been devoted to explore the right balance between recording overhead and debugging effectiveness, ranging from deterministic record-replay techniques [19, 17, 18, 13] to partial record-replay [6, 14]. However, production infrastructures are often reluctant to adopt any intrusive system recording approaches due to deployment and privacy concerns.

In this paper, we present *Insight*, a system that can infer the execution path of a failed service request *inside* the production environment *without* any intrusive system recording. We view Insight as a first-step failure inference tool for the developer to gain useful knowledge about *how* a service request fails in the production computing environment. Insight can significantly expedite the debugging process by narrowing down the scope of diagnosis from thousands of functions to a few of them. Moreover, the failure paths reproduced by Insight can be fed into a debugger (e.g., GDB) or a symbolic execution engine [10, 8] for further analysis.

The key idea of Insight is to perform *in-situ* failure path inference *inside* the production environment. The rationale behind our approach is that the production com-

[1] The online services considered in this paper refer to those request and response services such as a web server or a virtual machine (VM) reservation service in an infrastructure-as-a-service cloud.

puting environment provides many useful clues for us to perform failure inference more efficiently than offline approaches. Those clues include both *environment data* (e.g., input logs[2], configuration files, state of the faulty component, interaction with other production servers such as database query results) and *runtime outputs* (e.g., console logs, system call traces). Our experiments show that using environment data and runtime outputs can greatly *reduce the failure path search scope* and provide *important guidance* for us to find the correct failure path.

When a request failure is detected, Insight dynamically creates a shadow component of the faulty production server which produced the error message or was identified by an online server component pinpointing tool [11, 20, 27]. We detect a request failure by intercepting error messages or employing system anomaly detection tools [32, 15]. Since the production server is still alive during non-crashing failures, the shadow component can inherit the failure states of the faulty production server. Moreover, the shadow component allows us to decouple failure inference from the production operation. The production server can continue to process new requests without worrying about losing important diagnostic information. Our current prototype implements dynamic shadow component creation by augmenting the live virtual machine (VM) cloning technique [9, 22, 26]. Our scheme allows the shadow component to acquire environment data and runtime outputs from the production environment while imposing minimum disturbance to the production operation.

Insight proposes a novel *guided* binary execution exploration scheme that can efficiently leverage the production environment data and runtime outputs as guidance to search the failure paths. We make careful design choices in our failure inference algorithm in order to meet the following requirements: 1) *binary-only* since we cannot assume source code is available in the production environment; 2) *fast* path search in order to leverage the "fresh" environment data at the failure moment (i.e., the environment does not change much and the failure-triggering inputs or similar inputs are still in the buffer of the recent input log.); 3) no *intrusive* recording; and 4) support *both interpreted and compiled* programs.

Our guided binary execution exploration starts by replaying the last input in the input log when a failure is detected. However, Insight does not require the exact failure-triggering input to find the failure path since our binary execution exploration scheme can inherently handle incorrect environment data (e.g., different inputs, outdated or missing query results). During replay, we use the runtime outputs as guidance to stop searching along wrong paths, that is, if the replay produces a mismatched output, we roll back the execution to the previous branch point and flip the branch condition value (e.g., from `true` to `false`) to search a different path. If no matched path is found using the current input, we replay the next input in the input log and repeat the above process. We also support concurrent multi-path search to further shorten the failure path search time. Multi-path search also allows Insight to find multiple candidate failure paths that match the output of the production run.

We consider both *console log messages* and *system call traces* in the output matching. Most production servers already record console logs. If the production program produces many console log messages, our experiments show that Insight can rely on console log messages to produce high fidelity failure paths. However, if the production program includes very few console log messages, we propose to use system calls as hints to search paths between console logs. We chose to match system calls because they often represent key operations and can be collected using kernel-level tracing tools [5, 16] with low overhead ($< 1\%$).

We intentionally skip the constraint checking during the binary path search in order to achieve *fast* failure path inference in the production computing environment. With the help of environment data, we observe that Insight only needs to flip a small number of branches and the chance of finding an infeasible path is small. To filter out infeasible paths in our final result, we can apply constraint solver [12, 29, 24] to the candidate failure paths found by Insight, which is much faster than applying constraint solver during the path search.

We make the following contributions in this paper:

- We propose to perform *in-situ* failure path inference using a dynamically created shadow server *inside* the production computing environment.
- We present a *guided* binary execution exploration algorithm that can use available *environment data* (e.g., inputs, configuration files, states of interacting components) and *runtime outputs* (e.g., console logs, system calls) as guidance to quickly find the failure path over binary code directly.
- We evaluate Insight using real system failures. Experiments show that in-situ failure path inference is feasible. Insight can efficiently use the environment data and runtime outputs when they are present to find high fidelity failure paths within minutes.

The rest of the paper is organized as follows. Section 2 compares our work with related work. Section 3 presents the design and implementation of Insight. Section 4

[2] We observe that most production servers buffer a set of recent inputs. Although it might be impractical to assume the input log access for *offline* diagnosis (e.g., the privacy concern), it is easy to acquire the input log within the production computing environment.

presents the experimental results. Finally, the paper concludes in Section 5.

2 Related Work

Production-run failure debugging is a well known challenging task. In this section, we focus on reviewing the work that is most related to Insight and describing the difference between Insight and previous approaches.

Triage [33] first proposed an onsite production run failure diagnosis framework. It uses checkpoint-replay with input/environment modification to perform just-in-time problem diagnosis by comparing good runs and bad runs. Although Insight shares the same idea of onsite failure analysis with Triage, Insight differs from Triage in the following major ways. First, Triage performs onsite debugging on the production server directly, which can cause significant downtime to the online service. In contrast, Insight creates a shadow server to decouple the failure inference from the production operation. Second, Insight does not rely on repeated replays with input/environment modifications, which can incur a long failure analysis time and sometimes difficult to achieve in production systems. In comparison, Insight provides a fast binary execution exploration approach that uses the environment data and runtime outputs as guidance to search the failure paths on a dynamically created shadow component.

Alternatively, previous work (e.g., [19, 17, 18, 13, 30, 6, 28, 25, 39]) has proposed to introduce application-level or system-level instrumentation and infer the failure path based on instrumentation data. However, large-scale production computing environments are reluctant to adopt continuous intrusive system recording approaches due to overhead and deployment concerns. For example, *Aftersight* [13] proposed to decouple complex program analysis from normal executions using VM record and replay techniques. However, VM recording can impose high overhead to the normal production execution (e.g., worst case overhead reached 31% and 2.6x for some workloads [13]). Crameri et al. [14] proposed to use static and dynamic analysis to identify those branches that depend on input and only record those branches for failure reproduction. In comparison, Insight does not record any branch during the production run but instead exploits production environment data and runtime outputs to find the correct failure path onsite immediately after the failure occurs.

Another alternative is to perform *offline* failure inference using static source code analysis [37, 38]. For example, *Sherlog* [37] uses static source code analysis to infer the possible failure paths from console logs. *ESD* [38] uses program source code and bug reports (i.e., core dump information) to reproduce a failure execution. ESD first statically analyzes the source code to infer the control path capable of reaching the bug location, and then symbolically executes the program along the inferred control path to reproduce the failure execution. Because reproducing a production run failure outside the production environment is challenging [33], offline analysis cannot leverage the production environment data (e.g., inputs, configuration files, interaction results) or some runtime outputs that are difficult to obtain offline (e.g., system calls). Moreover, it is difficult for the offline approach to localize environment issues (e.g., network failure, wrong database query results). S^2E [12] provides an in-vivo multi-path analysis framework using selective symbolic execution over binaries for finding all potential bugs. In contrast, Insight aims at quickly finding the execution path for a specific occurring production-run failure. S^2E also does not consider runtime outputs when finding the failure path.

We view Insight as a first-step light-weight failure inference tool that can be used inside the production environment. We can apply the static/dynamic program analysis or symbolic execution to the candidate failure paths found by Insight to further validate the feasibility of the failure paths and localize root cause related branches.

3 System Design and Implementation

In this section, we describe the design and implementation details of the Insight system. We first present the dynamic shadow server creation scheme. We then describe our guided binary execution exploration algorithms.

3.1 Dynamic Shadow Server Creation

When a service request failure is detected, Insight dynamically creates a shadow component of the production server on a separate physical host using live VM cloning [9, 22]. Since Insight targets non-crashing failures and performs immediate cloning, we assume that the state of the shadow component is similar to the state of the production server when the failure occurs. We found this assumption holds for all the server failures we tested in our experiments.

The current prototype of Insight uses a pre-copy live KVM VM cloning system [26]. However, we can integrate Insight with other VM cloning techniques easily. Insight only requires a brief stop-and-copy phase (e.g., $<$ 100 milliseconds [26]) where the production component is paused temporarily for transferring any remaining dirty pages. During the stop-and-copy phase, the production server just pauses its processing but can continue to receive the user requests in its input buffer. For all the server systems we tested, Insight can complete the

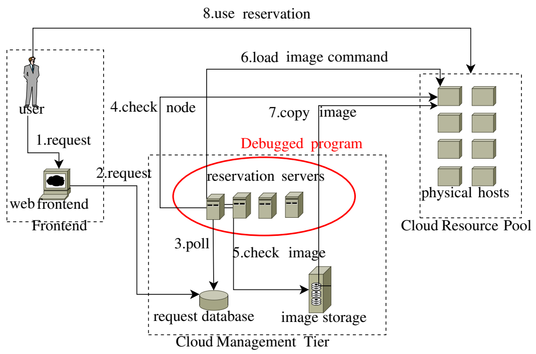


Figure 1: Our field study production server: VM reservation servers in the VCL cloud computing infrastructure [3]. The user makes a VM reservation request via a web interface. The request is stored in a database which is continuously polled by the reservation server. The reservation server forks a new process for handling each VM reservation request. First, the reservation server allocates a set of physical hosts for the user. If these hosts do not have the VM images required by the user, the reservation server then loads requested images from an image database. The reservation server then starts the `ssh` service and creates a user account for the user.

whole shadow component creation process within tens of seconds. Additionally, Insight performs transparent fast disk cloning to make the shadow component completely independent of the production server [26].

After the cloning is done, we need to reconfigure the shadow server to prepare it for the failure reproduction. Note that all the reconfigurations do not require any modification to the server software. Because live VM cloning makes the shadow server inherit all the state from the production server, which includes the IP address, the shadow server may immediately send out network packets using the same IP address as the production server, causing duplicate network packets and application errors. To avoid this, we first disconnect the network interface of the shadow server, clear the network buffer, and then reconnect the network interface of the shadow server with a new IP address.

To leverage the production environment for failure reproduction, we need to allow the shadow server to interact with other servers in the production environment for retrieving needed information. Figure 1 shows our field study production server which is a VM reservation server in an infrastructure-as-a-service cloud. The reservation server needs to interact with a MySQL database server to search for available physical hosts, look up the VM image name, and update the reservation state. Insight registers the shadow server with the database server using event-driven application auto-configuration [26]. Other interactions can be enabled in a similar way.

If the interaction requires the shadow server to read information from the environment (e.g., query from a database), the interaction is allowed. However, if the interaction requires the shadow server to update some information in the environment (e.g., write to a database), the interaction will be filtered to avoid undesired disturbance to the production server. We use an interaction filtering proxy to intercept outputs from the shadow server and drop selected outputs based on the query type. The proxy runs outside the shadow server software but on the same physical host with the shadow server. For example, our field study production server is written in Perl. We implemented the interaction filtering proxy within the Perl interpreter. We can also perform interaction recording on the *shadow server* to log important environment data which will be helpful for developers to diagnose a failure caused by an environment issue.

Insight is resilient to false alarms by providing light-weight runtime failure path inference and flexible cloning. If a false alarm is confirmed by the online anomaly detection tool before the shadow server is started, we simply cancel the live VM cloning operation. If a false alarm is confirmed after the shadow server is already started, we issue a delete command to the shadow server and release all resources allocated to the shadow server. In our field study server system, we use the critical error messages for detecting failures, which has few false positives [21]. We can also combine the error message detection with other failure prediction tools [31] to further reduce the false alarms.

3.2 Guided Binary Execution Exploration

Insight performs guided binary execution exploration in the shadow component to find the failure path. The execution exploration engine intercepts conditional jump statements (e.g., JZ, JNE, JE) in the binary code and explores different execution paths by manipulating the jump conditions (`true` or `false`). We assume all the conditional statements including the `switch` statements are translated into one or multiple conditional jump statements in the binary code. For example, in C/C++ program, we can compile the code using the `fno-jump-tables` option in `gcc`.

To start the execution exploration, we first replay the last input in the input log when the failure is detected. We employ an input replay proxy to retrieve the input log from the production server when the failure is detected. As mentioned in the Introduction, most production servers buffer recent inputs in an input log file. For example, a web server stores its input (i.e., HTTP requests) in the `access_log` file. For VCL reservation server, the inputs (i.e., VM reservation requests) are stored on a database server. Although our experiments show that inputs play a crucial role in the failure path inference, Insight does not require the exact failure-triggering input to find the failure path.

During the replay, we check whether the shadow component produces the same outputs (i.e., console log messages, system call sequences) as the failed service request. We will describe the output matching scheme details in the next section. A replayed path can produce mismatched outputs either because we did not replay the exact failure-triggering input or because some environment data (e.g., database content) was changed during the shadow component creation. We use an *unmatched output as a hint* to stop searching along a wrong path. Under those circumstances, the execution is rolled back to the previous branch point and we flip the branch condition to search a different path. If rolling back to the previous branch point still cannot produce any matched failure path, we rollback to the branch point before the previous branch point and so on. To avoid redundant search, we stop the rollback process when we see the previous console log message again. If no matched path is found using the current input, we replay the next input in the input log and repeat the above process. To support the above mechanism, Insight performs process checkpointing at each branch point and each console log output. We implement the process/thread checkpointing using $fork$.

Insight supports concurrent multi-path search to achieve fast failure reproduction. We implement the concurrent multi-path search by using a set of probing processes/threads called probes to explore different execution paths simultaneously. When the probe encounters a conditional jump statement, it forks a new child probe for exploring both the `true` and the `false` branches concurrently. To avoid overloading the system with a large number of concurrent searches, we set a concurrency quota *CQ* to limit the number of probes that can simultaneously run. When the number of probes exceeds CQ, we make the parent probe wait and allow the child probe to explore either the `true` or `false` branch. If the child probe produces an unmatched output, we kill the child probe to discontinue the search along the wrong path and release one concurrency quota. If the parent probe of the terminated child probe is waiting for the quota, the parent probe will be signaled to continue its exploration. When a probe produces the next matched output (i.e., console log message or system call), we stop the exploration and switch back to concrete execution mode (i.e., continue the execution without forking).

If an explored path contains a loop, Insight forks a new child probe at the beginning of each iteration by default. The parent probe will then exit the loop (i.e., the `false` branch) and allow the child probe to continue to execute the next iteration of the loop (i.e., the `true` branch). However, if the program does not produce any console log messages or system calls within the loop, Insight will never get any hint on when to stop exploring the loop. To avoid unnecessary loop explorations, Insight performs loop detection by checking for repeated program counters within one function. If no console log message or system call is produced within the loop, we disable exploration for that loop branch statement (i.e., do not fork new child probe) and let the loop exit naturally as its normal execution.

When a probe produces the same complete console log and system call sequences as the failed request, Insight marks the execution path explored by the probe as one *matched failure path*. Our approach can also find multiple matched failure paths simultaneously. The failure path inference will be terminated after the target number of matched failure paths are found or the search process times out. We also annotate each reproduced path with useful diagnostic information such as which branch points were manipulated by our exploration process and what the environment values were when the branch points were flipped by our system. Developers can use this information to decide the fidelity of the reproduced paths and perform informed value inferences.

Since Insight works on binaries directly, most Insight components can be applied to compiled or interpreted programs written in different languages without any modification. The only program-specific parts are how to intercept branch statements and change branch conditions. Insight currently supports Perl and C/C++ programs. For Perl programs, we modified the Perl interpreter to intercept the conditional jump statement. The jump condition value is stored in the interpreter's execution stack. We modify the jump condition value by changing the execution stack value. For C/C++ programs, Insight uses the Pin tool [23] to intercept the conditional jump statements and modify the jump conditions by changing the appropriate flags (i.e., jump flag, carry flag, overflow flag, and parity flag) in the EFLAGS register. Note that the above system modification and instrumentation are only applied to the shadow server during the execution exploration time. Insight does not perform any modification or instrumentation to the production server.

3.3 Runtime Output Matching

Insight uses runtime outputs as hints to check whether it explores a correct or incorrect path. We chose to match two different types of runtime outputs: console log messages [35, 37] and system calls for the following reasons. Production systems often produce console log messages for debugging production-run failures [35, 37]. In today's practice, console logs often provide the sole information source for diagnosing production run failures. Since console log messages are inserted by software developers for recording operations considered to be important, they are often able to provide useful clues about key program execution states. However, we also observe some systems (e.g., open source software) contain a limited number of log messages. Under those circumstances, we propose to match system call traces because system calls can be easily collected using kernel-level tracing tools with negligible overhead ($<$ 1% CPU load) and system calls often denote the key operations in the program. Different from user-level tracing tools such as ptrace [4], kernel-level tracing tools impose little overhead by avoiding context switches. We use SystemTap [5] in our current prototype.

During binary execution exploration, we continuously match the console log messages and system call sequences produced by the explored execution path with those from the failed production-run request. For console log matching, we adopt the same strategy as previous work [35, 37] by only considering the static text parts called *message templates* since the variable parts (e.g., timestamp, variable values) typically differs over different runs. Those message templates can be easily extracted from the source code and provided to Insight by software developers. Alternatively, we can extract the message templates directly from log files [34], which is however orthogonal to our work. We can also leverage parameter run-time values in the console log messages to extract more hints about the failure. We might be able to increase the failure path accuracy using those parameter values by incorporating Insight with taint analysis techniques. However, doing so will probably increase the runtime overhead. Our current results show that Insight can still successfully infer the failure paths without using those parameter values.

If the console log is too sparse, Insight still faces the challenge of large exploration scope. Thus, we use system calls as hints to guide our path search between any two consecutive console log messages L_1 and L_2. We observe that each console log message is written into the console log file using a sequence of `sys_write` system calls. The system call sequence S in-between those two groups of `sys_write` calls are marked as the system call sequence between L_1 and L_2. We use `readlink` and file descriptor contained in each `sys_write` to identify whether it writes into the console log file. When we perform failure path search between L_1 and L_2, we match the system call sequence S. We currently only consider system call types when we perform matching. We could also consider system call arguments or return values, which, however, might increase the system call tracing and matching overhead significantly.

When a mismatched system call is encountered, we roll back the exploration to the previous branch point and flip the branch condition to execute a different path. During our experiments, we observe that requiring an exact match sometimes prevents us from finding any matched path. The reason is that the same function call such as `malloc` might invoke slightly different numbers of system calls (e.g., `mmap`) depending on the application's heap usage. During those circumstances, we allow k mismatches (measured by string edit distance) to occur during system call sequence matching. We start from $k = 0$ and gradually increase k until we either find a matched path or our search times out. During our experiments, we find k needs to be no more than 2.

4 System Evaluation

In this section, we present the experimental evaluation for the Insight system. We first present our evaluation methodology followed by our experimental results.

4.1 Evaluation Methodology

Case study systems. We first test Insight using the virtual computing lab (VCL) [3] which is a production cloud computing infrastructure. VCL has been in production use for 9 years and has over 8000 daily users. Figure 1 shows the architecture of VCL. The key control part in VCL is the cluster of reservation servers which are written in about 145K lines of Perl code. The database server is configured to allow access from hosts in the same subnet, thus allowing the access from the shadow component. In our experiments, we deploy the Insight system on all the reservation servers and perform the in-situ failure path inference over the reservation servers which produce the reservation failure messages.

We also test Insight with several real software bugs in a set of open source softwares (Apache, Squid, Lighttpd, PBZIP2, aget, and GNU Coreutils).

Case study failures. We evaluated Insight using a set of real failures listed in Table 1. We also report the number of function calls and branch points contained in each failure execution path along with the root cause function of each failure. Each failure contains one error message. In our experiments, we detect failures by intercepting error messages: console log messages containing *critical* or *fatal* keyword or are written into *stderr*.

Evaluation metrics. We first evaluate whether the reproduced failure execution path is useful for debugging by checking whether the reproduced execution shows the same failure symptom (i.e., throwing out the same error messages), covers the root cause functions and branch statements. We then evaluate the precision and efficiency of Insight using the following metrics:

System name	**System description**	**LOC**	**Failure path length**		**Num. of console log msgs**	**Failure description**	**Root cause function**
			Num. of functions	**Num. of branches**			
VCL (v2.2.1)	VM reservation server	145K	112	378	132	**Overlapping reservation failure**: User tries to request the same VM reservation twice.	*computer_not_being_used*
VCL (v2.2.1)	VM reservation server	145K	299	1331	290	**Network failure**: The management node fails to create the VM reservation on a physical host due to the network failure on the host.	*_ssh_status*
VCL (v2.2.1)	VM reservation server	145K	298	1328	409	**Authentication failure**. The management node is unable to login into the reservation host due to a missing public key.	*run_ssh_command*
VCL (v2.2.1)	VM reservation server	145K	147	601	178	**Image corruption failure**. The VM image file corresponding to the user request is corrupted and cannot be copied.	*load*
Apache (httpd-2.0.55)	Web server	176K	176	21212	1	**Authentication failure**. Apache rejects a valid request due to incorrect file name setting for AuthGroupFile option (#37566).	*groups_for_user*
Apache (httpd-2.2.0)	Web server	209K	164	4983	1	**CGI failure**. Apache does not handle a malformed header generated by CGI script correctly (#36090).	*ap_scan_script_header_err_core*
Squid (v2.6)	Web cache and proxy server	110K	588	19679	195	**Non-crashing stop failure**. Squid is not able to handle a long value of "ACL name" option (#1702).	*aclParseAclList*
Lighttpd (v1.4.15)	Web server	38K	730	4308	3	**Proxy failure**. Lighttpd could not find the back-end server when configured as a reverse proxy for 1 back-end server with round-robin policy (#516).	*mod_proxy_check_extension*
PBZIP2 (v1.4.15)	Multithreaded data compression	3.9K	41	58	14	**Decompression failure**. The program fails to decompress files with trailing garbage (#886625).	*decompress ErrCheck Single*
aget (v0.4.1)	Multithreaded download accelerator	1.5K	2	8	1	**Download failure**. The program fails to download a file when setting the number of threads bigger than the maximum concurrent connection allowed in the server holding the file.	*http_get*
rmdir (v4.5.1)	GNU coreutils	0.2K	2	24	2	**Option failure**. The program does not handle trailing slashes with the "-p".	*remove _parents*
ln (v4.5.1)	GNU coreutils	0.6K	1	47	1	**Option failure**. The program does not handle "target-directory" correctly.	*do_link*
touch (v7.6)	GNU coreutils	0.5K	1	7	1	**Time failure**. The program rejects a valid input with the leap second.	*main*

Table 1: Real system failures used in our experiments. All the failures have one error log message produced during the failure run.

1) *Call path difference* denotes the deviation of the call path discovered by Insight from the original call path of the failed production run. The call path consists of a sequence of invoked functions during the execution. We used the *string edit distance* to measure the deviation between two compared call paths. We instrumented all the tested programs to record the original call path of the failed production run. Generally, the call path difference reflects how close the reproduced execution is to the original execution.

2) *Normalized branch difference*. We use the branch difference to denote the deviation at the branch level between the path reproduced by Insight and the original failure path. We also use the *string edit distance* to measure the branch difference between two execution paths. To perform comparison between different application failures, we normalize the branch difference of each failure using its maximum string edit distance between the reproduced path and the original path (i.e., no overlapping at all).

Generally, the call path difference and the branch difference reflect how close the reproduced execution is to the original execution. The branch difference is a more fine-grained comparison than the call path difference.

3) *Percentages of flipped branches* denotes the percentage of the branches whose conditions are manipulated by Insight due to incomplete environment information or different input.

4) *Exploration time* defines the time taken by Insight to discover the target number of the matched failure paths.

5) *Performance impact and overhead.* We evaluate the runtime performance impact of Insight to the production system by comparing the per-request processing time between with and without the Insight system. We also report the overhead of the Insight system in terms of additional resource consumptions.

Impact of environment data. To understand the impact of the environment information on the accuracy of our in-situ failure path inference, we compare the failure inference accuracy results under three different

Failure name	**Environment setting**	**Call path difference**	**Branch difference**	**Cover root cause functions**	**Cover root cause branches**
VCL overlapping reservation failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	0	Yes	Yes
	No environment data	0	0	Yes	Yes
VCL network failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	3.4%	Yes	Yes
	No environment data	Failed	Failed	N/A	N/A
VCL Authentication failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	2.7%	Yes	Yes
	No environment data	Failed	Failed	N/A	N/A
VCL Image corruption failure	Complete environment data	0	0	Yes	Yes
	Partial environment data	0	3.1%	Yes	Yes
	No environment data	Failed	Failed	N/A	N/A
Apache authentication failure	Original input	0	0	Yes	Yes
	Same input type + console log	17	66%	Yes	Yes
	Same input type + console log + system call	11	61.5%	Yes	Yes
Apache CGI failure	Original input	0	0	Yes	Yes
	Same input type + console log	140	41.8%	Yes	Yes
	Same input type + console log + system call	9	14.8%	Yes	Yes
Squid failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	0.0001%	Yes	Yes
	Same input type + console log + system call	0	0.0001%	Yes	Yes
Lighttpd failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	0.8%	Yes	Yes
	Same input type + console log + system call	0	0.8%	Yes	Yes
PBZIP2 failure	Original input	0	0	Yes	Yes
	Same input type + console log	1	4.4%	Yes	Yes
	Same input type + console log + system call	0	1.3%	Yes	Yes
aget failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	0	Yes	Yes
	Same input type + console log + system call	0	0	Yes	Yes
rmdir failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	17.5%	Yes	Yes
	Same input type + console log + system call	0	5.3%	Yes	Yes
ln failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	25.3%	Yes	Yes
	Same input type + console log + system call	0	9.1%	Yes	Yes
touch failure	Original input	0	0	Yes	Yes
	Same input type + console log	0	53.5%	Yes	Yes
	Same input type + console log + system call	0	0	Yes	Yes

Table 2: Summary of Insight failure path inference accuracy results.

environment contexts in the VCL system: 1) *complete environment data* where all the query results from the database are assumed to be the same during the whole failure path inference process; 2) *partial environment data* where all the database entries that are related to the failed reservation request are deleted to emulate the case when the failure inference is triggered after the reservation server clears up the failed requests. However, the shadow server can still access some general database information such as "computer load state" and "OS type" needed by the failure path finding; and 3) *no environment data* where all the query results from the database are unavailable. This emulates the case of offline failure reproduction.

Since all the C/C++ server failures are produced under stand-alone mode, we could not evaluate the impact of the environment data on the open source systems.

Impact of input. We evaluate the impact of the input by performing failure reproduction using the original failure triggering input or using a different input that does not trigger the failure but is of the same type as the original input.

We define the *same type of input* for different open source systems as follows: 1) *Apache authentication failure*: the same type of input is a http request to access a webpage with a correct AuthGroupFile setting; 2) *Apache CGI failure*: the same type of input is a http request to execute a normal CGI script; 3) *Squid failure*: we use a default configuration file with a normal "ACL name"; 4) *Lighttpd failure*: we use a http request using a reverse proxy with two back-end servers instead of one back-end server that makes the system fail; 5) *PBZIP2 failure*: we use a compressed file with no trailing garbage; 6) *aget failure*: we use a request that does not have restriction on the maximum concurrent connection; 7) *rmdir failure*: we use a command without the "-p" option; 8) *ln failure*: we use a command without the "target-directory" option; and 9) *touch failure*: we use an input that does not have a leap second.

Since the inputs (i.e., reservation requests) in the VCL system are stored in the database, they are considered as part of the environment data.

We conducted all the experiments on a computer cluster in our lab. Each cluster node is equipped with a quad-core Xeon 2.53GHz CPU, 8GB memory, and is connected to Gigabit network. Each host runs CentOS 6.2 64-bit with KVM 0.1.2. The guest VMs run CentOS 6.2 32-bit and are configured with one virtual CPU and 2GB memory. We repeated each experiment five times and report the mean and standard deviation values. In all experiments, we set the concurrency quota *CQ*=20.

4.2 Results and Analysis

Failure path accuracy result summary. Table 2 summarizes the accuracy of the failure paths reproduced by Insight for different failures. We observe that the environment data in VCL allows Insight to find the exact failure paths for all the VCL failures. With partial environment data, Insight can still achieve high accuracy with 0 call path difference and small ($<$ 5%) branch difference. However, when we remove all the environment data, emulating the offline failure reproduction situation, we cannot find any matched path for three out of four VCL failures after searching for several hours. This indicates that environment data plays a crucial role in timely failure path finding because they can greatly reduce the search scope for the binary execution exploration.

For the open source software systems, we observe that with the original failure-triggering inputs, Insight can always reproduce the exact failure path for each failure. When given the same type of input (see Section 4.1 for the definition of the same input type), Insight can still reproduce high fidelity failure paths with 0 call path difference and small ($<$ 10%) branch difference for most failure cases. The only exceptions are those failures that include only 1 error message without any other console log messages. This is expected as Insight has too few runtime outputs to guide the exploration. However, we observe that system call sequences can greatly help improve the failure path inference accuracy for the failure cases where sparse console logs are present. The branch difference reduction can be up to 100% (i.e., the branch difference of the touch time failure is reduced from 53.5% to 0).

We then validate whether the failure paths reproduced by Insight cover the root cause functions and branches by manually analyzing the source code. We observe that the failure paths found by Insight always cover the root cause functions and branches. Another interesting observation we observe is that the root cause branch points often do not appear right before the error message is produced, but reside in the middle of the execution path. For example, in the VCL overlapping reservation failure case, the error message "Reservation failed on vmsk1: process failed because computer is not available" does not provide the correct clue that the reservation failure is caused by an overlapping reservation not by the machine is not available. However, the failure path found by Insight covers the root cause branch where *pgrep* returns a process matching the request ID, indicating the same reservation has been made on the machine. For the Lighttpd failure, the reproduced path shows that the failure is caused by the back-end server lookup operation returning *empty* when the round-robin policy is employed and there is only one back-end server. The buggy code segment does not appear right before the error message.

We also examined how VM cloning helps Insight to find the failure paths. For example, the shadow component of the VCL reservation server inherits the configuration files that specify the supporting VM types (e.g., xCat, KVM), VM image locations, and public keys. Without those configuration parameters, it is extremely difficult to perform any replay. Similarly, the configuration file of Squid defines the permissions associated with the "ACL" name which are needed by

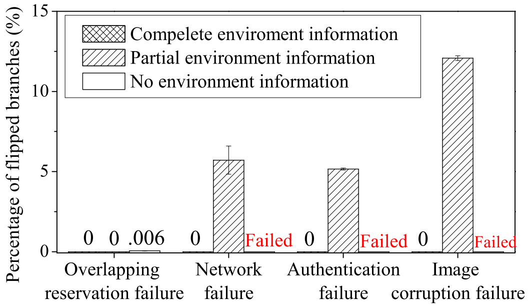


Figure 2: Percentage of flipped branches for VCL failures.

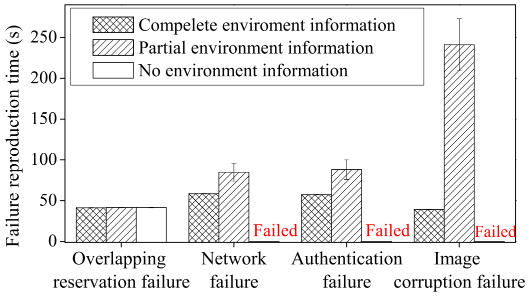


Figure 3: Total failure reproduction time (i.e., the shadow component creation time + failure path search time) for reproducing VCL failures.

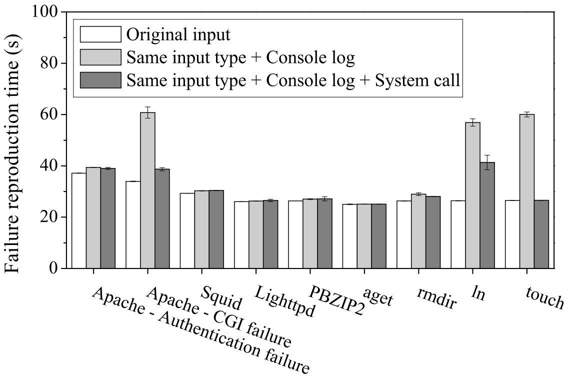


Figure 4: Failure reproduction time results for open source software bugs.

the failure reproduction. In Lighttpd, code modules such as mod_proxy and back-end servers are specified in the configuration file. Additionally, VM cloning also ensures the same third-party libraries are installed on the shadow component.

Detailed VCL failure reproduction results. We now present the detailed failure reproduction results for all the VCL failures. Table 2 shows the branch difference for different VCL failures. As mentioned in the accuracy result summary, Insight can find a failure path with little branch difference compared to the original failure path when in-situ failure reproduction is performed.

Figure 2 shows the percentage of branch points that are flipped by our binary execution exploration engine during path finding. We observe that Insight only flips a small number of branches when part of the environment data is not available.

Figure 3 shows the failure reproduction time for the VCL failures. The reproduction time includes the time for Insight to create the shadow component and the time taken to search the failure path. We observe that Insight can reproduce the failure path within a few minutes. It took Insight about 30 seconds to create the shadow component using live VM cloning. We also observe that the environment data has an impact on the failure reproduction time. When complete environment data is available, Insight can quickly reproduce the failure path within tens of seconds. When part of the environment data is missing, the reproduction time is longer, taking up to 250 seconds to complete. As mentioned before, when no environment data is available, Insight cannot find any matched failure paths after searching for several hours.

Detailed open source software failure reproduction results. We now present the results for the open source software bugs. Table 2 shows the normalized branch difference for the open source system failures. Figure 4 shows the failure reproduction time. We observe that with the original inputs, Insight can always reproduce the exact failure paths within tens of seconds including the shadow component creation time.

Given the same types of inputs, Insight can still reproduce the failure paths for Squid, PBZIP2, aget, and all the Coreutils failures within several minutes. However, Insight cannot reproduce the Apache authentication failure, the Apache CGI failure, and Lighttpd failure within a short period of time ($<$ 1 hour) which is a requirement for our in-situ failure reproduction. The reason is that those open source systems produce zero or very few (i.e., 3) console log messages except the error message during their failure executions. With such little guidance, Insight is faced with a large path search scope. Under those circumstances, Insight uses a code selector in a similar way as S^2E [12] to limit the path exploration within a specified target code module. For the Apache authentication failure, the target code module is the authentication module. For the Apache CGI failure, the target code module is the CGI component that handles CGI scripts. For Lighttpd, mod_proxy is the target code module. After limiting the path exploration scope, Insight is able to find the failure paths within tens of seconds.

We observe that system call sequences can greatly reduce the branch difference for those failures with few console logs. We also notice that the branch difference in the Apache authentication failure is significantly larger than the other failure cases when the original input is absent. The reason is that the program includes a large loop that includes many branch points but does not generate any console log message or system call. Because of the input difference, Insight executes the loop with a different number of iterations from the original failure execution, which causes the high branch difference.

Generally, we observe a higher branch differences in the open source failures than those in the VCL failures. This is expected as the open source software systems have less environment data to leverage under the stand alone mode and contain fewer console logs than the VCL system. Insight can definitely benefit from a rich set of environment data and a system with a good number of console logs. Based on our observation and feedback from our industry partners, we believe most production systems do contain abundant console logs as they are the sole information source for the software developer to diagnose production failures.

We also wish to compare Insight with existing static analysis and symbolic execution approaches. Unfortunately, none of them can support the Perl program that forms the main part of the VCL production cloud management service. For open source software systems, we found that Insight can achieve much faster

System	Production runtime overhead		Logging overhead (1 day)				Shadow creation time	Stop-and-copy time
	With system call tracing	With shadow component	Console log	Input log	Interaction log	System call log		
VCL	N/A	$< 0.3\%$	0.49 ± 0.01 GB	0.13 ± 0.01 GB	0.86 ± 0.01 GB	N/A	26.7 ± 2.3 s	49.6 ± 15.9 ms
Apache	$< 1\%$	$< 0.2\%$	0.3 ± 0.01 MB	19.6 ± 0.1 MB	N/A	11.9 ± 0.01 MB	23± 1.3 s	38.6 ± 6.5 ms

Table 3: Performance and resource overhead of the Insight system. Request rate in VCL: 120 VM reservation requests per minute. Request rate in Apache: 50 HTTP requests per second.

failure reproduction. For example, static analysis techniques need up to 28 minutes to analyze an Apache failure [37]. Symbolic execution requires up to 6 hours to explore a program with 1.3 KLOC [12]. This is expected as Insight can leverage many environment data and runtime outputs to greatly reduce the path search scope.

Insight system overhead. Table 3 shows the performance and resource overhead of the Insight system for the VCL reservation server and the Apache server. The results for other open source servers are omitted as they are similar to the Apache results. Insight does not require any system instrumentation during the production run except the system call tracing. We observe that the system call tracing imposes $<$ 1% performance impact and <1.5% CPU load to the production server. The performance impact is measured by comparing the per-request processing time when running systems without system call tracing and with system call tracing. We also measure the performance impact for the production operation when the production server runs concurrently with the shadow server. Again, we observe very little performance impact. We also study the logging overhead incurred by Insight. We can see the logging overhead is small compared to the capacity of modern storage systems. Finally, we measured the shadow component creation time and stop-and-copy time for different servers. The results show that we can finish the live VM cloning and shadow server configuration within 30 seconds. During the shadow server creation, we only need to pause the production server for less than 100 milliseconds.

5 Conclusion

We have presented Insight, an in-situ failure path inference system for online services running inside the production computing environment. Insight uses a shadow component to achieve efficient onsite failure inference while imposing minimum interference to the production service. Insight employs a guided binary execution exploration process to achieve accurate failure path inference by exploiting the production-site environment data and two different types of runtime outputs (i.e., console logs, system calls).

Our initial prototype implementation shows that Insight is both feasible and efficient. We tested Insight using real request failures collected on a production cloud computing infrastructure and a set of real software bugs in open source software systems. Our experiments show that Insight can efficiently use the environment data and runtime outputs to find the failure paths with high fidelity (i.e., little difference from the original failure path) within a few minutes. Insight is lightweight and unobtrusive, imposing negligible overhead to the production service.

Acknowledgment

We thank the anonymous reviewers and our shepherd Erik Riedel for their valuable comments. We also thank VCL system administrators Aaron Peeler and Andy Kurth for providing us with the log data and their generous help on validation. We thank Anwesha Das for helping with the experiments. This work was sponsored in part by NSF CNS0915567 grant, NSF CNS0915861 grant, NSF CAREER Award CNS1149445, U.S. Army Research Office (ARO) under grant W911NF-10-1-0273, IBM Faculty Awards and Google Research Awards. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF, ARO, or U.S. Government.

References

[1] The 10 biggest cloud outages of 2012. http://www.crn.com/slide-shows/cloud/240144284/the-10- biggest-cloud-outages-of-2012.htm/.

[2] Amazon EC2 Service Disruption Summary. http://aws.amazon.com/message/65648/.

[3] Apache VCL. https://vcl.ncsu.edu.

[4] Process trace. http://linux.die.net/man/2/ptrace/.

[5] Systemtap. https://sourceware.org/systemtap/.

[6] G. Altekar and I. Stoica. ODR: output-deterministic replay for multicore debugging. In *SOSP*, 2009.

[7] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *OSDI*, 2012.

[8] G. Brat, K. Havelund, S. Park, and W. Visser. Java pathfinder-second generation of a java model checker. In *Workshop on Advances in Verification*, 2000.

[9] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via VM state coloring. In *EuroSys*, 2011.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[11] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI*, 2004.

[12] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, 2011.

[13] J. Chow, T. Garnkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX ATC*, 2008.

[14] O. Crameri, R. Bianchini, and W. Zwaenepoel. Striking a new balance between program instrumentation and debugging time. In *Eurosys*, 2011.

[15] D. Dean, H. Nguyen, and X. Gu. UBL: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In *ICAC*, 2012.

[16] M. Desnoyers and M. R. Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *Linux Symposium*, 2006.

[17] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: global comprehension for distributed replay. In *NSDI*, 2007.

[18] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *USENIX ATC*, 2006.

[19] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: an application-level kernel for record and replay. In *OSDI*, 2008.

[20] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *SIGCOMM*, 2009.

[21] K. Kc and X. Gu. ELT: efficient log-based troubleshooting system for cloud computing infrastructures. In *SRDS*, 2011.

[22] H. A. Lagar Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. SnowFlock: rapid virtual machine cloning for cloud computing. In *EuroSys*, 2009.

[23] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[24] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[25] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[26] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

[27] H. Nguyen, Z. Shen, Y. Tan, and X. Gu. Fchain: Toward black-box online fault localization for cloud systems. In *ICDCS*, 2013.

[28] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. PRES: probabilistic replay with execution sketching on multiprocessors. In *SOSP*, 2009.

[29] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *FSE*, 2005.

[30] D. Subhraveti and J. Nieh. Record and transplay: partial checkpointing for replay debugging across heterogeneous systems. In *SIGMETRICS*, 2011.

[31] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *PODC*, 2010.

[32] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *ICDCS*, 2012.

[33] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: diagnosing production run failures at the user's site. In *SOSP*, 2007.

[34] R. Vaarandi. Mining event logs with slct and loghound. In *NOMS*, 2008.

[35] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, 2009.

[36] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *SOSP*, 2011.

[37] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ASPLOS*, 2010.

[38] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Eurosys*, 2010.

[39] J. Zhou, X. Xiao, and C. Zhang. Stride: search-based deterministic replay in polynomial time via bounded linkage. In *ICSE*, 2012.

Automating the choice of consistency levels in replicated systems

Cheng Li[‡], João Leitão[†], Allen Clement[‡], Nuno Preguiça[†], Rodrigo Rodrigues[†], Viktor Vafeiadis[‡]
[‡] *MPI-SWS* [†] *NOVA Univ. of Lisbon / CITI / NOVA-LINCS*

Abstract

Online services often use replication for improving the performance of user-facing services. However, using replication for performance comes at a price of weakening the consistency levels of the replicated service. To address this tension, recent proposals from academia and industry allow operations to run at different consistency levels. In these systems, the programmer has to decide which level to use for each operation. We present SIEVE, a tool that relieves Java programmers from this error-prone decision process, allowing applications to automatically extract good performance when possible, while resorting to strong consistency whenever required by the target semantics. Taking as input a set of application-specific invariants and a few annotations about merge semantics, SIEVE performs a combination of static and dynamic analysis, offline and at runtime, to determine when it is necessary to use strong consistency to preserve these invariants and when it is safe to use causally consistent commutative replicated data types (CRDTs). We evaluate SIEVE on two web applications and show that the automatic classification overhead is low.

1 Introduction

To make web services more interactive, the providers of planetary-scale services—such as Google, Amazon, or Facebook—replicate the state and the application logic behind these services either within a data center or across multiple data centers, and direct users to a single (and preferably the closest or least loaded) replica [11, 28, 14]. Gaining performance through replication, however, comes at a price. To avoid the high cost of coordinating among replicas, the infrastructures that provide replicated services resort to weak consistency levels such as causal consistency [23, 4], eventual consistency [11], or timeline consistency [10]. Under these weak consistency models, good performance is extracted by the fact that only a small number of replicas needs to be contacted for the execution of each operation before producing a reply to the user. However, this adaption also modifies the semantics provided by the replicated service, when compared to strong consistency models like serializability [36] or linearizability [13], where a replicated system behaves like a single server that serializes all operations. Using weak consistency models requires special care, because their semantics may violate user expectations, for example by allowing an auction service to declare two different users to be the winners of the same auction.

Recognizing this tension between performance and meeting user expectations, many research [18, 30, 20, 33] and commercial [31, 12, 25] systems offer the choice between executing an operation under a strong or a weak consistency model. All of these proposals require the application programmer to declare which operations should run under which consistency level. In most cases this is done explicitly by extending the interface for operation execution with an indication of the desired consistency level, while in a recent proposal this is done implicitly by associating a consistency SLA to each operation, ranking and assigning utility values to the various consistency levels [33]. The problem with these strategies is that they impose on the application programmer the non-trivial burden of understanding the semantics of each operation and how the assignment of different consistency levels to different operations influences overall semantics that are perceived by the users.

In this paper, we address this problem by automating the process that assigns consistency levels to the various operations, focusing on an important and widely deployed class of applications, namely Java-based applications with a database backend. To achieve this goal, we build on prior work [20] that defines sufficient properties for safely using a weak consistency model (namely causal consistency), and changes the replication model to separate the generation of the side effects of an operation from their application to the state of the replicas. Adapting existing applications to this model requires manual work that can be challenging and error-prone. First, one must transform every application operation into a generator and a commutative shadow operation. Second, one must correctly identify which shadow operations may break some application invariant, and label them appropriately so that they execute under strong consistency.

In order to ease the burden on the programmer, we have designed SIEVE, a tool that automates this adaptation. Using SIEVE, we require the programmer to only specify the application invariants that must be preserved and to annotate a small amount of semantic information about how to merge concurrent updates. SIEVE achieves this automation by addressing the two identified challenges using the following approach:

First, to ensure convergence under weak consistency, SIEVE automatically transforms the side effects of every application operation into their commutative form. To this end, we build on previous work on commutative replicated data types (CRDTs) [29, 26], i.e., data types whose concurrent operations commute, and apply this concept to relational databases. This allows programmers to only specify which particular CRDT semantics they intend by adding a small annotation in the database schema, and SIEVE automatically generates the shadow operation code implementing the chosen semantics.

Second, SIEVE uses program analysis to identify commutative shadow operations that might violate application-specific invariants when executed under weak consistency semantics, and runs them under strong consistency [20]. To make the analysis accurate and lightweight, we divide it into a potentially expensive static part and an efficient check at runtime. The static analysis generates a set of abstract forms (*templates*) that represent the space of possible shadow operations produced at runtime, and identifies for each template a logical condition (*weakest precondition*) under which invariants are guaranteed to be preserved. This information is then stored in a dictionary, which is looked up and evaluated at runtime, to determine whether each shadow operation can run under weak consistency.

We evaluate SIEVE using TPCW and RUBiS. Our results show that it is possible to achieve the performance benefits of weakly consistent replication when it does not lead to breaking application invariants without imposing the burden of choosing the appropriate consistency level on the programmer, and with a low runtime overhead.

2 Background

Before presenting the various aspects of SIEVE, we first introduce the system model it builds upon, and the operation classification methodology it relies on.

In previous work [20], we defined RedBlue consistency, where operations can be labeled red (strongly consistent) or blue (weakly consistent). Red operations are totally ordered with respect to each other, meaning that they execute in the same relative order at all replicas, and therefore no two red operations execute concurrently. (This corresponds to the requirements of serializability.) In contrast, blue operations can be reordered with respect to other operations, provided they preserve causality (corresponding to causal consistency).

A pre-requisite to being able to label operations as blue is that operations should commute, so that executing them in a different order at various replicas does not lead to a divergent replica state. To increase the space of commutative operations, we proposed a change in the state machine replication model such that operations are split between a generator operation running only on the replica that first receives the operation and producing no side effects, and a shadow operation sent to all replicas, which effectively applies the side effects in a commutative way. More formally, in the original state machine replication model, an operation u deterministically modifies the state of a replica from S to S' (denoted as $S + u = S'$). In the proposed model, the application programmer decomposes every operation u into generator and shadow operations g_u and $h_u(S)$, respectively, where S is the replica state against which g_u was executed. The pair of generator and shadow operations must satisfy the following correctness requirement: for any state S, $S + g_u = S$ and $S + h_u(S) = S + u$.

Given this system model, we defined sufficient conditions for labeling operations in a way that ensures that application invariants are not violated. In particular, a shadow operation can be labeled blue if it commutes with all other shadow operations, and it is *invariant safe*, meaning that if states S and S' preserve the invariants, then the state $S' + h_u(S)$ does so as well.

3 Overview

Using RedBlue consistency requires the programmer to generate commutative shadow operations and identify which can be blue and which must be red. Our goal is to automate these two tasks, to the extent possible.

For the first task, we leverage the rich commutative replicated data type (CRDT) literature [29, 26], which defines a list of data types whose operations commute. CRDTs can be employed to produce commutative shadow operations that converge to identical final states, independent of the order in which they are applied. Shadow operations are thus constructed as a sequence of updates to CRDT data types that commute by construction.

The challenge in developing shadow operations based on CRDTs is that the programmer must explicitly transform the applications to replace all the application state mutations by calls to the appropriate CRDT object. This involves not only identifying the parts of the programs that encode these actions, but also understanding the catalogue of CRDT structures and choosing the appropriate one. To minimize this programmer intervention, we focus on two-tier architectures that store all of the state that must persist across operations in a database. This gives us two main advantages: (1) We can automatically identify the actions that mutate the state, namely the operations that access the database. (2) We can reduce the user intervention to small annotations referring to the database data organization.

The second challenge SIEVE addresses is automatically labeling commutative shadow operations. To this end, for each shadow operation that is generated, we need to decide whether it is invariant safe, according to the definition in Section 2. (Commutativity does not need to be checked since the previous step ensures that shadow operations commute by design.) To automate the classification process, two design alternatives that represent two ends of a spectrum: (1) a dynamic solution, which determines at runtime, when the shadow operation is produced, whether that shadow operation meets the invariant safety property, and (2) a fully static solution that determines which combinations of initial operation types, parameters, and initial states they are applied against lead to generating a shadow operation that is invariant safe. The problem with the former solution is that it introduces runtime overheads, and the problem with the latter solution, as we will detail in Section 5, is that the static analysis could be expensive and end up conservatively flagging too many operations as strongly consistent.

To strike a balance between the two approaches, we split the labeling into a potentially expensive static part and a lightweight dynamic part. Statically, we generate a set of templates corresponding to different possible combinations of CRDT operations that comprise shadow operations, along with weakest preconditions for each template to be invariant safe. Then, at runtime, we perform a simple dictionary lookup to determine which template the shadow operation falls into, so that we can retrieve the corresponding weakest precondition and determine whether it is met.

These two main solutions lead to the high level system architecture depicted in Figure 1. The application programmer writes the *application code* as a series of transactions written in Java, which access a database for storing persistent state. Beyond the application code, the only additional inputs that the programmer needs to provide are *CRDT annotations* specifying the semantics for merging concurrent updates and a set of application-specific invariants. The static analyzer then creates *shadow operation templates* from the code of each transaction, where these templates represent different sequences of invocations of functions in a *CRDT library*. The analyzer also computes the *weakest preconditions* required for each template to be invariant safe.

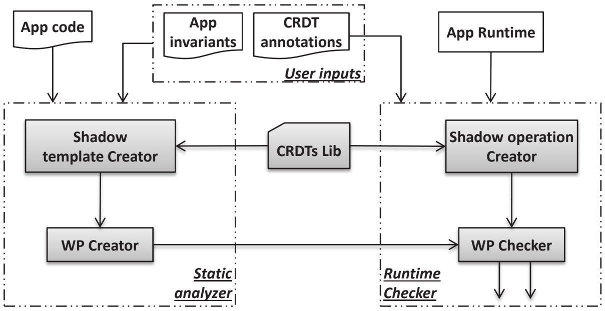


Figure 1: Overview of SIEVE. Shaded boxes are system components comprising SIEVE. (WP stands for weakest precondition.)

At runtime, application servers run both the Java logic and the *runtime checker*, and interact with a database server (not shown in the figure) and the replication tier (not shown in the figure). While executing a transaction, the application server runs the generator operation inside a *shadow operation creator*, which, instead of directly committing side effects to the database, generates a shadow operation consisting of a sequence of invocations from the CRDT library. This shadow operation is then fed to the *weakest precondition checker* to decide which static template it falls into, and what is the precondition required for the operation to be invariant safe, which allows the runtime to determine how to label the operation. The labeled shadow operation is then fed to the replication system implementing multi-level consistency. In the following sections we further discuss the design and implementation of the main components of this architecture.

4 Generating shadow operations

This section covers how we automate the conversion of application code into commutative *shadow operations*.

4.1 Leveraging CRDTs

We leverage several observations and technologies to achieve a sweet spot between the need to capture the semantics of the original operation when encoding its side effects and the desire to minimize the amount of programmer intervention. First, we observe that many applications are built under a two-tier model, where all the persistent state of the service is stored in a relational database accessed through SQL commands. Second, we leverage CRDTs [26], which construct operations that commute by design by encapsulating all side effects into a library of commutative operations.

SQL type	CRDT	Description
FIELD*	LWW	Use last-writer-wins to solve concurrent updates
	NUMDELTA	Add a delta to the numeric value
TABLE	AOSET, UOSET, AUSET, ARSET	Sets with restricted operations (add, update, and/or remove). Conflicting ops. are logically executed by timestamp order.

Table 1: Commutative replicated data types (CRDTs) supported by our type system. * FIELD covers primitive types such as integer, float, double, datetime and string.

These two concepts allow us to achieve commutativity while overcoming the disadvantage of CRDTs, namely the need to adapt applications. This is because the state of two-tier applications is accessed through the narrow SQL interface, and therefore we can focus exclusively on adapting the implementation of SQL commands to access a CRDT. For example, database tables can be seen as a set of tuples, and therefore all the calls in the original operation to add or remove tuples in a table can be replaced in the shadow operation with a CRDT set add or remove, which, in turn, is implemented on top of the database. The programmer only has to select the appropriate merging strategy (i.e., the adequate CRDT type) to encode these operations, without being required to program these CRDT transformations or to change the code of each operation.

However, it is impossible to completely remove the programmer from the loop, due to the choice of which CRDT to use for encoding appropriate merging semantics. For instance, when an integer field of a tuple is written to in a SQL update command, the programmer could have two different intentions in terms of what the update means and how concurrent updates should be handled: (1) the update can represent a delta to be added or subtracted from the current value (e.g., when updating the stock of a certain item), in which case all concurrent updates should be applied possibly in a different order at all replicas to ensure that no stock changes are lost, or (2) it can be overwriting an old value with a new value (e.g., when updating the year of birth in a user profile), in which case an order for these updates should be arbitrated, and the last written value should prevail. Even though both strategies ensure convergence, their semantics differ significantly. For example, the second strategy leads to a final state that does not reflect the effects of all update operations.

Since the appropriate merging strategy is application-specific, the programmer has to convey this decision. To minimize this input, we only require the programmer to declare such semantics on a per-table and per-attribute basis. In more detail, we provide programmers a number of CRDT types (shown in Table 1). These types form two categories: field, which is the smallest component of a record and defines its commuting update operation in the presence of concurrency, and set, which is a collection of such records plus the support for commutative appending or removing. Programmers only need to annotate the data schema with the desired CRDT type using the following annotation syntax: $@[CRDTName][TableName|DataFieldName]$

```
@AUSET CREATE TABLE exampleTable (
    objId INT(11) NOT NULL,
    @NUMDELTA objCount INT(11) default 0,
    @LWW objName char(60) default NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB
```

Figure 2: Annotated table definition schema.

Figure 2 presents a sample annotated SQL table creation statement. We assign `exampleTable` the type `AUSET` (Append-Update Set), a CRDT set that only allows append and update operations, thus precluding the concurrent insertion and deletion of the same item (less restrictive CRDT sets also exist). The field `objCount` associated with `NUMDELTA` always expects a delta value to be added or subtracted to its current value. By default, if no annotations are provided, we conservatively mark the corresponding table or field to be read-only.

4.2 Runtime creation of shadow operations

With these schema annotations in place, it is easy to generate commutative shadow operations at runtime. The idea is to invoke the original operation upon the arrival of a new user request (as would happen in a system that does not make use of shadow operations) but with the difference that all the calls to execute commands in the database are intercepted by a modified JDBC driver that builds the sequence of CRDT operations that comprise the shadow operation as the original operation progresses. Furthermore, using the schema annotations, SIEVE maps each database update to an appropriate merge semantics and replaces the operations on a certain table with the appropriate operations over the corresponding CRDT type.

For instance, to create a shadow operation for a transaction that updates `objCount` in Figure 2, when an update is invoked, we first query the old value s, and then, given the new value s', we compute a `delta` by subtracting s from s'. Finally, we use `delta` and the primary key *pk* of the corresponding object to parameterize a CRDT operation that reads the tuple identified by *pk* and then adds `delta` to it.

Finally, when the original operation issues a commit to the database, the tool outputs a shadow operation containing the accumulated sequence of CRDT operations.

```
Begin transaction;
for(int i = 0; i < x.length; i++){
  if(x[i] < 100)
    x[i]++;
  else
    x[i] = -100            }
End transaction;
```

(a) Original code

```
func txnShadow(int[] obsX, int[] deltaA){
    for (i = 0; i < obsX.length; i++){
        if(obsX[i] < 100)
            CRDT_x[i].applyDelta(deltaA[i]);
        else:
            CRDT_x[i].applyDelta(deltaA[i]); }
}
```

(b) Possible corresponding shadow template

Figure 3: Code snippet of a transaction and a possible template for the corresponding shadow operation.

5 Classification of shadow operations

In this section we explain how we automatically label shadow operations as strongly or weakly consistent.

5.1 Overview

As mentioned in Section 3, a possible solution would be to statically compute the combinations of operation types, parameters, and initial states that generate invariant safe shadow operations. This can be done by performing a weakest precondition computation—a common technique from Programming Languages and Verification research for which some tool support already exists—which enables us to statically compute, given the code of each operation and the application-specific invariants (which are inserted as postconditions), a precondition over the initial state and operation parameters that ensures the invariant safety property. However, this raises the following two important problems.

First, there is a scalability problem, which is exemplified by the following hypothetical code for the generator operation, assuming an invariant that the state variable x should be non-negative. (For simplicity, we write conventional Java code accessing variable x instead of SQL.)

```
void generator(string s) {
  if (SHA-1(s)==SOME_CONSTANT) {
     if (x>=10)
        x -= 10;
  } else
     x +=10;
}
```

The problem with this code is that a weakest precondition analysis to determine which values of s lead to a negative (non-invariant safe) delta over x is computationally infeasible, since it amounts to inverting a hash function. As such, we would end up conservatively labeling the shadow operations generated by this code as red (i.e., the weakest precondition would be FALSE). Even though this is an extreme example, it highlights the difficulty in handling complex conditions over the input, even when the side effects are simple. In particular, that there are only three patterns of side effects produced by this generator, regardless of the inputs provided to the generator operation. Based on this observation, to simplify the weakest precondition computation and to minimize the space of strongly consistent shadow operations, our static analysis is conducted over the set of possible sequences of CRDT operations that can be generated, which is the same as saying that we analyze all possible shadow operations. We call each possible sequence of shadow operations that can be generated by a given generator operation a template. In the above example, there are only three sequences of shadow operations that can be generated: the empty sequence, adding a delta of 10, and adding a delta of -10. From these three possible sequences, only a delta of -10 leads to a weakest precondition of FALSE, i.e., is always non-invariant safe. The remaining ones have a weakest precondition of TRUE.

The second challenge that needs to be overcome is related to handling loops. The generator code in Figure 3(a) illustrates that the number of iterations in the loop can be unbounded, which in turn leads to an unbounded number of CRDT operations in the shadow operation. To abstract this, we could produce a template that preserves the loop structure, such as the one in Figure 3(b). However, when computing a weakest precondition over this piece of code, verification tools face a scalability problem, which is overcome by requiring the programmer to specify loop invariants that guide the computation of this weakest precondition [17]. Again, this would represent an undesirable programmer intervention.

To address this challenge, we note that in many cases (including all applications that we analyzed), loop iterations are independent, in the sense that the parts of the state modified in each iteration are disjoint. Again, this is illustrated by the example in Figure 3, where the loop is used to iterate over a set of items, and each iteration only modifies the state of the item being iterated.

This iteration independence property enables us to significantly simplify the handling of loops. In particular, when generating the weakest precondition associated with a loop, we only have to consider the CRDT operations invoked in two sets of control flow paths, one where the code within the loop is never executed, and another with all possible control flow paths when the loop is executed and iteration repetitions are eliminated. (We will explain in detail how to handle loops using an example

Sequential path	Description
$2 \cdot 3 \cdot 4 \cdot 2$	only if
$2 \cdot 3 \cdot 6 \cdot 2$	only else
$2 \cdot 3 \cdot 4 \cdot 2 \cdot 3 \cdot 6 \cdot 2$	else follows if
$2 \cdot 3 \cdot 6 \cdot 2 \cdot 3 \cdot 4 \cdot 2$	if follows else

Table 2: Distinct sequential paths obtained for the transaction in Figure 3(a).

in the following subsection.) This condition can then be validated against each individual iteration of the loop at runtime and, given the independence property, this validation will be valid for the entire loop execution.

In our current framework, the iteration independence property is validated manually. In all our case-study applications, it was straightforward to see that this property was met at all times. We leave the automation of this step as future work.

5.2 Generating templates

Instead of reasoning about the generator code, our analysis is simplified by reasoning about the side effects of each code path taken by the generator operation. Furthermore, we can cut the number of possible code paths by eliminating code sections that are repeated due to loops.

To perform this analysis, we require an algorithm for extracting the set of sequential paths of a transaction and eliminating loop repetition. The high level idea of this algorithm is to split branch statements and replace loops with all non-repeating combinations of branches that can be taken within a loop. The algorithm works as follows. First, for every transaction, we create its path abstraction, which is a regular expression encoding all control flow information within that transaction. In the example shown in Figure 3(a), its path abstraction is $2 \cdot (3 \cdot (4|6) \cdot 2)*$, where numbers represent the statement identifiers shown in the figure, $\cdot$ concatenates two sequential statements, $|$ is a binary operator that indicates that the statements at its two sides are in alternative branches, and $*$ represents repetition within a loop. Second, we recursively apply the following two steps to simplify a path abstraction until it is sequential (i.e., no $*$ and $|$). For a path abstraction containing $*$, we create two duplicated abstractions, where one excludes the entire loop, and the other simplifies the loop into its body. For a path abstraction containing the operator $|$, we create two duplicated path abstractions, where one excludes the right operand and the other excludes the left operand. Additionally, if such $|$ is affected by a $*$, then we have to create another path abstraction combining both alternatives, i.e., where the if and the else sides are executed sequentially.

In the previous example, the set of sequential paths that is produced is shown in Table 2. By ignoring the read-only path where the loop is not executed, we only consider four cases, namely only the if or the else path, and the two sequences including both if and else. Because of the loop independence property, these cases are able to capture all relevant sequences of shadow operations. Note that we would only require considering one of the two orderings for the if and the else code within the loop, since their side effects commute, but taking both orderings into account simplifies the runtime matching of an execution to its corresponding path.

Given a set of sequential paths for a transaction, creating shadow operation templates become straightforward. For each path, we collect a sequence of statements specified by the identifiers in the abstraction from the corresponding control flow graph. Then, we translate every database function call into either a CRDT operation by following the instructions stated in Section 4, or a no-op operation (for read-only queries). Finally, all these CRDT operations are packed into a function, which denotes the shadow operation template. These CRDT operations are parameterized by their respective arguments, and the static analysis computes a weakest precondition over these arguments for the template to be invariant safe.

The final output from the static analysis is a dictionary consisting of a set of $< key, value >$ pairs, one for each previously generated shadow operation template, where *key* is the unique identifier of the template, and *value* is the weakest precondition for the template. The unique identifier of the template encodes the set of possible paths using signatures of CRDT operations in a restricted form of regular expression.

5.3 Runtime evaluation

Template/shadow operation matching. At runtime, it is necessary to evaluate the weakest precondition to classify operations as red or blue. To this end, we must lookup in the dictionary created during the static analysis the template corresponding to each shadow operation as it is produced.

The challenge with performing this lookup is that it requires determining the identifier of the shadow operation corresponding to the path taken, and this must be done by taking into account *only* the operations that are controlled by the runtime, i.e., the CRDT operations. This explains why the dictionary keys consist only of CRDT operations. With the shadow operation identifier, matching the path taken at runtime with the keys present in the dictionary is done efficiently by using a search tree.

Weakest precondition check. Finally, once the weakest precondition for the template that corresponds to a particular shadow operation is retrieved, we evaluate that precondition against the CRDT parameters of the shadow operation. This is achieved by simply replacing the variables in the precondition with their instantiated values and evaluating the final expression to either true or

App	Invariants
TCPW	$\forall item \in item\_table.\ \ item.stock \geq 0$
RUBiS	$\forall item \in item\_table.\ \ item.stock \geq 0$
	$\forall u,v \in user\_table.$ $u.uname = v.uname \implies u = v$

Table 3: Application-specific invariants

false. If the weakest precondition is evaluated to true the shadow operation is labeled blue, otherwise the shadow operation in labeled red.

After this step, the shadow operation is delivered to the replication layer, which replicates it using different strategies according to its classification.

6 Evaluation

In this section, we report our experience with implementing SIEVE, adapting existing web applications to run with SIEVE, and evaluating these systems.

6.1 Implementation

We implemented most of our tool using Java (15k lines of code), and changed parts of the Jahob code to obtain weakest preconditions in OCaml (553 lines of code). The backend storage system we used was a MySQL database. We used an existing Java parser [1] to parse java files. Finally, we connected our tool to the Gemini replication and coordination system [20] to enable both consistency classification and operation replication.

6.2 Use cases

To adapt an application to use SIEVE, one has to annotate the corresponding SQL schema with the proper CRDT semantics, specify all invariants, and finally the original JDBC driver must be replaced by the driver provided by SIEVE, to enable SIEVE to intercept interactions between the application and the database.

We applied SIEVE to two web application benchmarks, namely TPCW [9] and RUBiS [7]. Both of them simulate an online store and the interactions between users and the web application. There are two main motivations for selecting these use cases: (1) both have been widely used by the community to evaluate system performance; and (2) both have application-specific invariants that can be violated under weak consistency. (In our prior work [20] a social application is evaluated, but it made no sense to include this application because it did not contain any invariants that could be violated under weak consistency.)

For TPCW, we use `AOSET`, `AUSET`, `UOSET` and `ARSET` to annotate the database tables, no annotations for unmodified attributes, `NUMDELTA` for `stock`, and `LWW` for the remaining attributes. For RUBiS, we annotate its tables with `AUSET` and `AOSET`. We use `NUMDELTA` as annotations for both `quantity` and `numOfBids`, and no annotations or `LWW` for the remaining attributes. Identified invariants in these two applications are summarized in Table 3. For additional details, we refer the interested reader to the code available in [2].

In terms of the time required to do this adaptation, we do not report results for TPCW as we relied on this use case during the design and development phase of SIEVE. However for the RUBiS use case, the entire process was concluded in only a few hours. An interesting point to highlight is that SIEVE is able to detect inconsistencies between these annotations, enabling programmers to correct mistakes such as type omissions in the SQL schema that are inconsistent with the CRDT annotations.

In both our prior work [20] and the current work, the effort we made analyzing application code to determine invariants and merge semantics is unavoidable. In our prior work, however, we additionally spent a significant amount of time manually implementing merge semantics, and classifying shadow operations by taking into account their properties, for every application. SIEVE eliminates all this manual work, and limits human error.

6.3 Experimental setup

All reported experiments were obtained by deploying applications on a local cluster, where each machine has 2*6 i7 cores and 48GB RAM, and runs Linux 3.2.48.1 (64bit), MySQL 5.5.18, Tomcat 6.0.35, and Java 1.7.0.

6.4 Experimental results

Our experimental work aims at evaluating both the static analysis component of SIEVE and also the runtime component, which includes a performance comparison between each application using our tool, its unmodified version, and its version under RedBlue consistency where the entire classification is done manually and offline.

Concerning the static analysis component we focus on the following main questions: (*i*) How long does the static analysis process take to complete? (*ii*) What is the scalability of the static analysis component in relation to the size of the code base?

For the runtime component of SIEVE we focus on the following main questions: (*i*) Is the runtime classification of shadow operations accurate? (*ii*) What is the (runtime) overhead for adapted applications compared to their stand-alone unmodified counterparts? (*iii*) What are the performance gains obtained through weakly consistent replication using SIEVE?

6.4.1 Static analysis

As mentioned before, taking the application source code and CRDT annotations as input, SIEVE first maps each transaction into a set of distinct paths, and automatically transforms each path into a shadow operation template.

Transaction name	#paths	#templates	Transaction name	#paths	#templates	Transaction name	#paths	#templates
read-only txns (13)	1	0	createNewCustomer	2	2	doBuyConfirm-A	32	32
createEmptyCart	1	1	adminUpdate	4	4	doCart	36	36
refreshSession	1	1	doBuyConfirm-B	16	16			

Transaction name	#paths	#templates	Transaction name	#paths	#templates	Transaction name	#paths	#templates
ViewUserInfo	6	0	PutComment	10	0	PutBid	14	0
BrowseRegions	5	0	StoreComment	11	3	StoreBid	17	5
BuyNow	7	0	ViewBidHistory	11	0	AboutMe	37	0
SearchItemsByRegion	20	0	StoreBuyNow	13	6	RegisterItem	59	24
SearchItemsByCategory	20	0	BrowseCategories	13	0			
ViewItem	10	0	RegisterUser	14	3			

Table 4: Number of reduced paths and templates generated for each transaction in TPCW (top) and RUBiS (bottom).

Table 4 summarizes the number of paths (excluding loops) and the corresponding number of shadow operation templates that were produced by SIEVE for both TPCW and RUBiS. For TPCW, 15 out of the total 20 transactions only exhibit a single path, as the code of these transactions is sequential. The two most complex transactions in this use case are `doBuyConfirm` and `doCart`, which are associated with the user actions of shopping and purchasing. In contrast, most transactions in RUBiS have a more complex control flow, which generated a larger number of possible execution paths.

Note that the majority of transactions in both use cases do not lead SIEVE to produce any template. This happens when the transactions are read-only, and therefore do not have side effects. Additionally, in TPCW every path in an update transaction generates a shadow operation template, since system state is always modified. However, this is not true in RUBiS, because its code verifies several conditions, some of which lead to a read-only transaction.

As depicted in Table 5, the execution of SIEVE generated a total of 92 and 41 shadow operation templates for TPCW and RUBiS, respectively. In addition to these templates, our tool also generates automatically a set of Java classes that represent database data structures, which are necessary for computing weakest preconditions.

Table 6 depicts a full list of the different weakest preconditions generated by SIEVE for both use cases. These weakest preconditions alongside their respective shadow operation template identifiers are used by the runtime logic to classify shadow operations as either blue or red. A weakest precondition denoted by `True` implies that any shadow operation associated with that template is always invariant safe and therefore labeled blue. In contrast, a weakest precondition denoted by `False` implies that shadow operations associated to that template must always be classified as red. The remaining non-trivial conditions must be evaluated at runtime by replacing their arguments with concrete values. For instance, when a `doBuyConfirm` transaction produces a negative delta, then the condition will be evaluated to `False` and the corresponding shadow operation will be classified as red, otherwise the condition will be evaluated to `True` and the shadow operation will be classified as blue.

App	#code	templates		#db code	#specs
		num	#code		
TPCW	8.3k	92	1554	879	730
RUBiS	9.8k	41	251	477	371

Table 5: Overview of the output produced by the static analysis. "db code" refers to the Java classes representing database structures required for computing weakest preconditions.

	WP	Comments
TPCW	`True`	Not influencing invariants
	`delta` ≥ 0	Non-negative stock
RUBiS	`True`	Not influencing invariants
	`False`	Nickname must be unique
	`delta` ≥ 0	Non-negative quantity
	`quantity` ≥ 0	Non-negative quantity (new item)

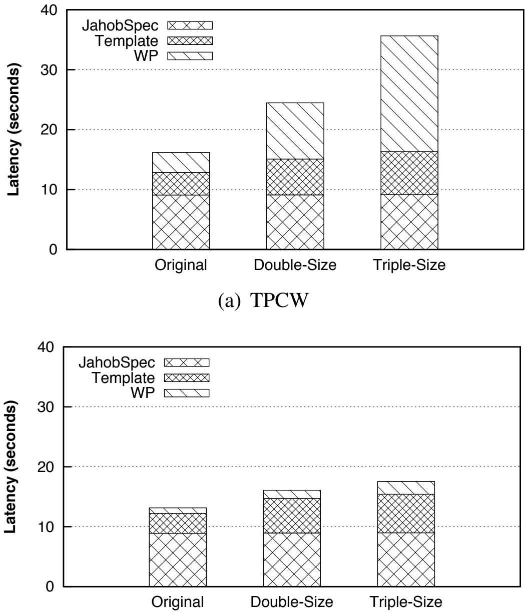
Table 6: Weakest preconditions (WP)

App	JahobSpec	Template	WP	Total
TPCW	9.1 ± 0.1	3.8 ± 0.1	3.3 ± 0.1	16.2 ± 0.3
RUBiS	8.9 ± 0.0	3.3 ± 0.3	0.9 ± 0.1	13.2 ± 0.3

Table 7: Average and standard deviation of latency in seconds for static analysis tasks (5 runs).

Cost of static analysis. A relevant aspect of the static analysis component in SIEVE is the time required to execute it. To study this we have measured the time taken by the static analysis and present the obtained results in Table 7. We not only measured the end-to-end completion time, but also the time spent for each step, namely, creating database data structures required by Jahob (JahobSpec), template creation (Template), and weakest precondition computation (WP). Overall, we can see that the execution time of the static component of SIEVE is acceptable, as less than 20 seconds are required to analyze both TPCW and RUBiS. The code generation phase including both JahobSpec and Template dominates the overall static analysis. Compared to TPCW, the time spent computing weakest preconditions is shorter in RUBiS, due to the smaller number of templates in Table 5.

Scalability. The code base size of TPCW and RUBiS is somewhat small when compared to deployed applica-



(a) TPCW

(b) RUBiS

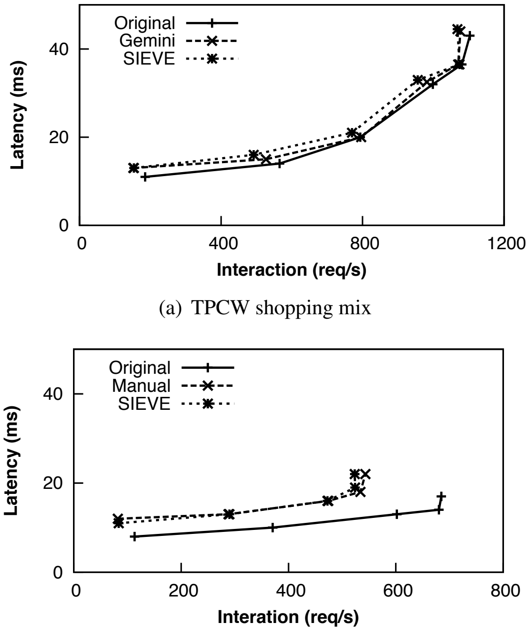
Figure 4: Static analysis time vs. code base size.

tions. This raises a question concerning the scalability of the static analysis component of SIEVE with respect to the size of the code base. In order to analyze this aspect of SIEVE we have artificially doubled and tripled the size of each application code base and measured the time spent analyzing these larger code bases when compared with the original. The results are shown in Figure 4. The time spent generating the data structures required by Jahob is constant, since we did not change the database schema. However, the time spent computing the weakest preconditions for templates in TPCW grows exponentially, and the time taken for the remaining steps presents a sub-linear increase. These results lead us to conclude that the static analysis of SIEVE may scale to reasonable code sizes, especially taking into account that this process is executed a single time when adapting an application through the use of SIEVE.

6.4.2 Runtime logic

We evaluated the runtime performance of our example applications using SIEVE on top of Gemini, which is a coordination and replication layer supporting generator and shadow operation execution [20].

Configurations. We populated the dataset for TPCW using the following parameters: 50 EBS and 10,000 items. For RUBiS we populated the dataset with 33,000 items for sale, 1 million users, and 500,000 old items. We exercised all TPCW workloads, namely browsing mix, shopping mix, and ordering mix, where the purchase activity varies from 5% to 50%. For RUBiS, we ran the bidding mix workload, in which 15% of all user activities generate updates to the application state.



(a) TPCW shopping mix

(b) RUBiS bidding mix

Figure 5: Throughput-latency graph without replication

Correctness validation. To verify that SIEVE labels operations correctly for both case studies, we inspected the log files generated by running SIEVE with TPCW and RUBiS, and we found that SIEVE conducts the same classification that was achieved manually in our previous work [20].

SIEVE runtime overhead. Next we compared the performance (throughput vs. latency) of the two applications across three single-site deployments: 1) SIEVE, 2) Original—the original unreplicated service without any overheads from creating and applying shadow operations, and 3) Manual—the RedBlue scheme with all labeling performed offline by the programmer. The expected sources of overhead for SIEVE are: *i*) the dynamic creation of shadow operations; and *ii*) the runtime classification of each shadow operation. The results in Figure 5 show that the performance achieved by SIEVE is similar to the one obtained with a manual classification scheme, and therefore the overheads of runtime classification are low. The comparison with the original scheme in a single site shows some runtime overhead due to creating and applying shadow operations (which is required for a replicated deployment so that all operations commute).

To better understand the sources of overhead imposed by SIEVE we measured the latency contribution of each runtime step executed by SIEVE and compared it with the latency contribution of these steps when relying on a manual adaptation. In particular, we focused on the following tasks: generator execution (producing a shadow operation), classification (determining shadow operation colors), and shadow execution (applying shadow opera-

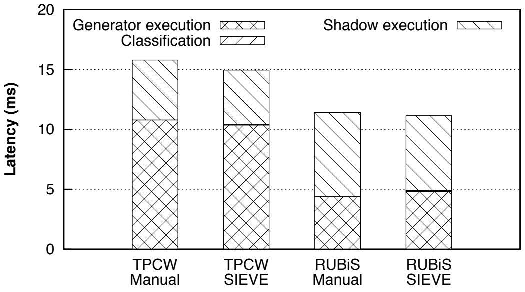


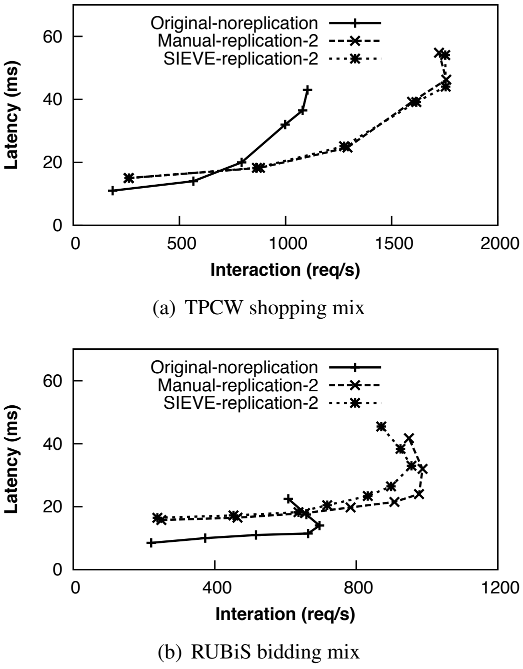
Figure 6: Breakdown of latency.

tions).

Figure 6 shows the average contribution to request latency of each of these steps (Only update requests are considered since read-only queries do not generate side effects.) For the manual adaptation, there is no latency associated with classifying shadow operations, since the classification of all shadow operations is pre-defined. In contrast, SIEVE performs a runtime classification, but the results show that the time consumed in this task is negligible. In particular, SIEVE takes 0.064 ± 0.002 *ms* and 0.072 ± 0.001 *ms* for looking up the dictionary and evaluating the condition for TPCW and RUBiS, respectively. Regarding the generator execution and shadow execution, both the manual adaptation and SIEVE present the same latency overheads.

Replication benefits. The results previously discussed in this section have shown that the use of SIEVE imposes a small overhead when compared to a standalone execution of the unmodified use cases, mostly due to runtime classification. However, SIEVE was designed to allow replication to bring performance gains through the use of weak consistency in replicated deployments. To evaluate these benefits, we conducted an experiment where we deployed the two applications (1) without replication, (2) using manual classification in Gemini, and (3) using SIEVE, with two replicas in the same site for the last two options. (The use of single site replication instead of geo-replication makes our results conservative, since the overheads of runtime classification become diluted when factoring in cross-site latency.)

The results in Figure 7 show that weakly consistent replication for a large fraction of the operations brings performance gains. In particular, one observes that the peak throughput with 2 replicated Gemini instances running TPCW is improved by 59.0%, and the peak throughput for RUBiS in this setting is improved by 37.4%. The additional latency introduced in this case is originated by the necessity of coordination among replicas to totally order red shadow operations. The results also confirm that the overhead of runtime classification when compared to the manual, offline classification are low. Note that there is a point where the throughput goes down while there is still an increase in latency in Figure 7(b).



(a) TPCW shopping mix

(b) RUBiS bidding mix

Figure 7: Throughput-latency graph with two replicas.

This happens because the database becomes saturated at this point.

7 Related work

We summarize and compare previous work with SIEVE according to the following categories:

Eventual consistency and commutativity. A large number of replicated systems have relied on eventual consistency for supporting low latency for operations by returning as soon as an operation executes in a single replica. These systems must handle conflicts that may arise from concurrent operations. In some systems, such as Bayou [34], Depot [24], and Dynamo [11], applications must provide code for merging concurrent versions. Other systems, such as Cassandra [19], COPS [22], Eiger [23] and ChainReaction [4], use a simple last-writer-wins strategy for merging concurrent versions. This simple strategy may, however, lead to lost updates.

Some systems have explored using operation commutativity to guarantee that all replicas converge to the same state, regardless of operation execution order. For example, Walter [32] includes a single pre-defined data type with commutative operations, *cset*. This system could be extended for supporting other data types with commutative operations proposed in the literature [26, 29]. Lazy replication [18] and RedBlue [20] support unordered execution of commutative operations defined by programmers. Furthermore, RedBlue [20] extends the space of commutative operations by decoupling operation generation and application, requiring only that operation application code is commutative.

Unlike these systems, SIEVE automatically adapts applications so that commutativity is obtained without modifying existing application code or adopting a new programming model – a commutative operation that encodes the operation side effects is automatically generated from the application code.

Multi-level consistency. As some application operations cannot execute correctly under eventual consistency, a few multi-level consistency models that combine eventual and strong consistency have been proposed [32, 20, 18, 33]. The properties of these models overlap with each other, and differ mainly in the composition of the different consistency levels. For instance, some work [32, 20] has found that it is sufficient to categorize operations into strong and weak consistency. Some other work [33] presents a more fine-grained division for read-only operations, which includes consistent prefix read, monotonic reads, and so on. We build on these models, and, in order to keep our design and our presentation simple, we follow the two-level consistency model proposed by RedBlue consistency [20].

Classification for multi-level consistency. In order to help developers adopt different proposals for multi-level consistency models, their creators introduced a few instructions to guide how to use their work. Relying on a probabilistic model, consistency rationing [16] associates different consistency levels with different states, instead of operations, and allows states to switch from one level to another at runtime. Unlike this approach, we partition operations into strong and eventual consistency groups. Both RedBlue consistency [20] and I-confluence [6] define conditions that operations must meet in order to run under weak consistency, i.e., without coordination. We build on this line of work and extend it so that an automatic tool, and not the programmer, is responsible for determining whether the operations meet these conditions.

To free programmers from the classification process, some researchers have attempted to apply program analysis techniques to reason about the consistency requirements of real applications. Alvaro et al. [5] identify code locations that need to inject coordination to ensure consistency, while Zhang et al. [36] inspect read/write conflicts across all operations. However, they focus on commutativity, and ignore application invariants, which are very important and taken into account by our solution. Very recently, Roy et al. [27] devised a way to summarize transaction semantics to allow sites to execute transactions independently, and without leading to inconsistencies. Their approach differs in the way that invariants are maintained, which resembles the concept of warranties [21] in that some operations cannot proceed so that others execute locally. Furthermore, the paper focuses extensively on the analysis of transaction code to determine their abstract semantics, which is complementary to the goal of our work since we rely on Jahob to determine only certain properties of shadow operations.

Commutativity and classification beyond eventual consistency. Commutativity has been explored in other settings to improve performance and scalability – e.g. in databases [35] and in OS design for multi-core systems [8]. Program analysis techniques have also been used to identify commuting code blocks. Aleen et al. [3] proposed a new approach to find commutative functions automatically at compile time for allowing legacy software to extract performance from many-core architectures. Kim et al. [15] used the Jahob verification system to determine commuting conditions under which two operations can execute in different orders. Unlike these two prior solutions that focus on identifying commutative code blocks, our tool automatically transforms operations by decoupling operation generation and application, which makes more operations commute [20], and we also focus on determining invariant safety.

8 Conclusion

In this paper we presented SIEVE, the first system to automate the choice of consistency levels in a replicated system. Our system relieves the programmer from having to reason about the behaviors that weak consistency introduces, only requiring the programmer to write the system invariants that must be preserved and provide annotations regarding merge semantics. Our evaluation shows that SIEVE labels operations accurately, incurring a modest runtime overhead when compared to labeling operations manually and offline.

Acknowledgments

The authors wish to express their gratitude to the anonymous reviewers, Rose Hoberman, and our shepherd Jinyang Li, whose comments improved the quality of the paper, as well as Thomas Wies for his assistance with Jahob. The research of R. Rodrigues has received funding from the European Research Council under an ERC starting grant. Authors from NOVA were supported by FCT/MCT project PEst-OE/EEI/UI0527/2014. Computing resources for this work were supported by an AWS in Education Grant.

References

[1] The web page of javaparser. `http://code.google.com/p/javaparser`. Accessed May-2014.

[2] SIEVE example files. `http://www.mpi-sws.org/~chengli/atc2014files/`, 2014.

[3] ALEEN, F., AND CLARK, N. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proc. of the 14th ASPLOS* (2009).

[4] Almeida, S., Leitão, J., and Rodrigues, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of the 8th ACM EuroSys* (2013).

[5] Alvaro, P., Conway, N., Hellerstein, J., and Marczak, W. R. Consistency analysis in bloom: a calm and collected approach. In *Proc. of the CIDR* (2011).

[6] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Coordination-avoiding database systems. *CoRR abs/1402.2237* (2014).

[7] Cecchet, E., and Marguerite, J. Rubis: Rice university bidding system. `http://rubis.ow2.org/`, 2009.

[8] 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 *Proc. of the 24th ACM SOSP* (2013).

[9] consortium, T. Tpc benchmark-w specification v. 1.8. `http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf`, 2002.

[10] 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. *In Proc. of the VLDB Endow. 1*, 2 (Aug. 2008).

[11] 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 *Proc. of the 21st ACM SOSP* (2007).

[12] Google. Welcome to google app engine. `https://appengine.google.com/`. Accessed May-2014.

[13] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990).

[14] Hoff, T. Latency is everywhere and it costs you sales - how to crush it. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`, 2009.

[15] Kim, D., and Rinard, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proc. of the 32nd ACM PLDI* (2011).

[16] Kraska, T., Hentschel, M., Alonso, G., and Kossmann, D. Consistency rationing in the cloud: Pay only when it matters. *In Proc. of the VLDB Endow. 2*, 1 (Aug. 2009).

[17] Kuncak, V. *Modular Data Structure Verification*. PhD thesis, EECS Department, MIT, 2007.

[18] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992).

[19] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010).

[20] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making georeplicated systems fast as possible, consistent when necessary. In *Proc. of the 10th USENIX OSDI* (2012).

[21] Liu, J., Magrino, T., Arden, O., George, M. D., and Myers, A. C. Warranties for faster strong consistency. In *Proc. of the 11th USENIX NSDI* (2014).

[22] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. of the 23rd ACM SOSP* (2011).

[23] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proc. of the 10th USENIX NSDI* (2013).

[24] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust. In *Proc. of the 9th USENIX OSDI* (2010).

[25] Oracle. Oracle NoSQL database. `http://www.oracle.com/us/products/database/nosql/overview/index.html`. Accessed May-2014.

[26] Preguica, N., Marques, J. M., Shapiro, M., and Letia, M. A commutative replicated data type for cooperative editing. In *Proc. of the 29th IEEE ICDCS* (2009).

[27] Roy, S., Kot, L., Foster, N., Gehrke, J., Hojjat, H., and Koch, C. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR abs/1403.2307* (2014).

[28] Schurman, E., and Brutlag, J. Performance related changes and their user impact. Presented at velocity web performance and operations conference, 2009.

[29] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proc. of the 13th SSS* (2011).

[30] Singh, A., Fonseca, P., Kuznetsov, P., Rodrigues, R., and Maniatis, P. Zeno: Eventually consistent byzantine-fault tolerance. In *Proc. of the 6th USENIX NSDI* (2009).

[31] Sivasubramanian, S. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proc. of the ACM SIGMOD* (2012).

[32] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proc. of the 23rd ACM SOSP* (2011).

[33] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. Consistency-based service level agreements for cloud storage. In *Proc. of the 24th ACM SOSP* (2013).

[34] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM SOSP* (1995).

[35] Weihl, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.* (1988).

[36] 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 *Proc. of the 24th ACM SOSP* (2013).

Sirius: Distributing and Coordinating Application Reference Data

Michael Bevilacqua-Linn, Maulan Byron, Peter Cline, Jon Moore, and Steve Muir

{Michael_Bevilacqua-Linn,Peter_Cline2,Jon_Moore}@comcast.com,
{Maulan_Byron,Steve_Muir}@cable.comcast.com
Comcast Cable

Abstract

The main memory of a typical application server is now large enough to hold many interesting *reference datasets* which the application must access frequently but for which it is not the system of record. However, application architectures have not evolved to take proper advantage. Common solutions based on caching data from a separate persistence tier lead to error-prone I/O code that is still subject to cache miss latencies. We present an alternative library-based architecture that provides developers access to in-memory, native data structures they control while neatly handling replication and persistence. Our open-source library *Sirius* can thus give developers access to their reference data in single-node programming style while enjoying the scaling and robustness of a distributed system.

1 Introduction

Many applications need to use *reference data*—information that is accessed frequently but not necessarily updated in-band by the application itself. For example, a TV guide application may need to know the titles, seasons, and descriptions of various TV shows. Furthermore, many reference datasets now fit comfortably in memory, especially as the exponential progress of Moore's Law has outstripped these datasets' growth rates. To illustrate, the total number of feature films listed in the Internet Movie Database (IMDb) grew 40% from 1998 to 2013 [11], whereas commodity server RAM grew by 12,000% in the same period.[1]

In addition, common "*n*-tier" service architectures tend to separate the management of reference data from the application code that must use it. In order to maintain low latency, application developers maintain caching layers, either built into a supporting library such as an object-relational mapper (ORM) or managed explicitly with external caches like *memcached* [6] or *Redis* [17]. These schemes may be difficult to use correctly [18], force developers to deal with I/O, and are still subject to cache misses which drive up upper-percentile latencies. This raises the question at hand:

> How can we keep this reference data entirely in RAM, while ensuring it gets updated as needed and is easily accessible to developers?

In this paper, we describe an alternative architecture to the classic *n*-tier approach for managing reference data, based on a distributed system library called *Sirius*. Sirius offers the following combination of properties:

- Simple and semantically transparent interface and freedom to use arbitrary, native data structures for the in-memory representation of the reference data (Section 4).
- Eventually consistent, near real-time replication of reference data updates across datacenters connected by wide-area networks (WANs) and designed for robustness to certain common types of server and network failures. (Section 5).
- Persistence management and automated recovery after application server restarts (Section 6).
- Adequate write throughput to support reference data updates (Section 7).

We have been using Sirius for over 15 months in production to power applications serving tens of millions of clients. We discuss the operational aspects of this architecture in Section 8.

2 Background

We will begin with a description of *reference data* to establish the context for this paper. As a motivating example, we will use the domain of professionally-produced

[1] 1998 Sun Ultra 2 server with 256 MB vs. 2013 Elastic Compute Cloud (EC2) "m3.2xlarge" instance with 30 GB.

television and movie metadata—the primary use case that motivated the design and implementation of Sirius. Examples of this metadata include facts such as the year *Casablanca* was released, how many episodes were in Season 7 of *Seinfeld*, or when the next episode of *The Voice* will be airing (and on which channel). Such reference datasets have certain distinguishing characteristics:

Small. The overall dataset is not "big data" by any stretch of the imagination, and in fact can fit in main memory of modern commodity servers. For example, the entertainment metadata we use is in the low tens of gigabytes (GB) in size.

Very high read/write ratio. Overwhelmingly, this data is write-once, read-many—for example, the original *Casablanca* likely won't get a different release date, *Seinfeld* won't suddenly get new Season 7 episodes, and *The Voice* will probably air as scheduled. However, this data is central to almost every piece of functionality and user experience in relevant applications—and those applications may have tens of millions of users.

Asynchronously updated. End users largely have a read-only view of this data and hence are not directly exposed to the latency of updates. For example, an editorial change to correct a misspelling of "*Cassablanca*" may take a while to propagate to all the application servers, with end users seeing the update occur but without being able to distinguish whether it was accomplished in minutes or milliseconds. There *are* thresholds of acceptable latency, however: if a presidental press conference is suddenly called, schedules may need to be updated within minutes rather than hours.

Such reference datasets are relatively common: a download of the U.S. Federal Reserve's data on foreign exchange rates is 1.2 MB compressed [4], the entire *Encylopædia Britannica* is 4.2 GB [27], and the collection of global daily weather measurements from 1929–2009 is only 20 GB [2]. In fact, the most recent versions of all the English Wikipedia articles total around 43 GB in uncompressed XML format as of December 2, 2013, which already fits in the 88 GB of RAM on a "cr1.8xlarge" EC2 instance. In other words, many reference datasets are already "small," and more will become so as server memory sizes continue to grow.

2.1 Operational Environment

We would like to access our reference data in the context of providing modern interactive, consumer-facing Web and mobile application services. This imposes some important constraints and design considerations on a potential solution. In particular, these services must support:

Multiple datacenters. We expect our services to run in multiple locations both to minimize latency to geographically disparate clients but also to protect against datacenter failures caused by the proverbial backhoe or regional power outages. For example, AWS has experienced multiple total-region failures but no multi-region or global failures to date.

Low access latency. Because we are building interactive applications where actual humans are waiting for responses from our application servers, we must have fast access to our reference data. Service latencies directly impact usage and revenue [9].

Continuous delivery. Our services will be powering products that are constantly being evolved. We expect our application developers to spend a lot of time modifying the code that interacts with their reference data, and we expect to be able to deploy code updates to our production servers multiple times per day. In order to do this safely, we prefer approaches that support easy and rapid automated testing.

Robustness. Since we will be supporting large deployments, we expect to experience a variety of failure conditions, including application server crashes, network partitions, and failures of our own service dependencies. We would like our overall system to continue operating—although perhaps with degraded functionality—in the face of these failures.

Operational friendliness. Any system of sufficient complexity will exhibit emergent (unpredictable) behavior, which will likely have to be managed by operational staff. We would like our approach to have a simple *operational interface*: it should be easy to understand "how it works," things should fail in obvious but safe ways, it should be easy to observe system health and metrics, and there should be "levers" to pull with predictable effects to facilitate manual interventions.

3 Approach

As we have seen, we can fit our reference data comfortably in RAM, so we start with the idea that we will simply keep a full mirror of the data on each application server, stored in-process as native data structures. This offers them ultimate convenience:

- No I/O calls are needed to external resources to access the data. Correspondingly, there is no connection pool tuning required nor is there a need to handle network I/O exceptions.

- Automated tests involving the reference data can be vanilla unit tests that neither perform I/O nor require extensive use of mock objects or test doubles.

- Profilers are actually useful for finding and fixing application bottlenecks since behavior is isolated from external dependencies. By contrast, common "*n*-tier" application servers are often I/O bound rather than CPU bound.

- Developers have full freedom to choose data structures directly suited to the application's use cases.
- There are no "cache misses," as the entire dataset is present; access is fast and predictable.

Of course, this approach raises several important questions in practice. How do we keep the mirror up to date? How do we run multiple servers while ensuring each gets every update? How do we restore the mirrors after application server restarts?

3.1 Update publishing

We assume there are external systems responsible for curating the reference data set(s), and that these system will *publish* updates to our server rather than having our server poll the system of record looking for updates. This event-driven approach is straightforward to implement in our application; we update the native data structures in our mirror and continue serving client requests. We model this interface after HTTP as a series of PUTs and DELETEs against various URL keys.

Furthermore, this introduces a separation of operational concerns: should the external system of record become unavailable, our application simply stops receiving updates without needing the error handling code that would be required by a polling approach. Indeed, our application cannot (and does not need to) distinguish a failed source system from one with no fresh updates.

Finally, this is a parsimonious way to process updates where much of the reference data does not change from moment to moment, but where there *is* a regular trickle of updates. For example, our production datasets experience a nominal update rate of no more than 150 updates per second. This incremental approach allows us to avoid re-downloading and re-parsing a full mirror and then either calculating diffs ourselves or swapping out a mirror entirely for an updated version.

3.2 Replication and Consistency

To run a cluster of application servers, we need to apply the updates at every server. To isolate the system of record from needing to know how many application servers are deployed, we route updates through a load balancer, then rely on the servers to replicate the updates to each other. Thus, a source system can publish each update once. This also isolates the system of record from individual server failures.

Because we will be operating a distributed system across a WAN, we know we will experience frequent network partitions due to unusually high latency, route flaps, or other failures. Therefore, the CAP theorem dictates we will need to decide between availability and consistency during these times. We need read access to the reference data at all times and will have to tolerate some windows of inconsistency. That said, we want to preserve at least *eventual consistency* to retain operational sanity, and we can tolerate some unavailability of *writes* during a partition, as our reference data updates are asynchronous from the point of view of our clients.

To achieve this, our cluster uses a variant of the Multi-Paxos [5] protocol to agree on a consistent total ordering of updates, and then have each server apply the updates in order. This means that the system of record cannot publish updates without a quorum of available servers, but it also allows servers to read the reference data without coordination. A formal consensus protocol also allows us to consistently order updates from multiple systems of record. We provide more detail in Section 5.

Finally, we use a *catchup protocol* for filling in lost or missing updates from peers. This protocol can also be used to set up dependent servers that do not participate in Paxos but nonetheless receive and apply the stream of updates. As a result, we can set up very flexible replication topologies to scale out read access to the dataset.

3.3 Persistence

As with many Paxos implementations, each server provides persistence by maintaining a local transaction log on disk of the committed updates. When a server instance starts up, it replays this transaction log to rebuild its mirror from scratch, then rejoins the replication protocol described above, which includes "catch up" facilities for acquiring any missing updates.

Because updates are modeled as idempotent PUT and DELETE operations, we can compact this log to retain only the most recent updates for active keys, without requiring the application to implement checkpointing. Our live log compaction scheme is described in Section 6.

3.4 Library Structure

Finally, we have structured the overall system around a library called Sirius that handles the Paxos implementation, persistence, log compaction, and replay. The hosting application is then conceptually separated into two pieces: its external interface and business logic, and its mirror, as shown in Figure 1. There are then two different data paths used for processing reference data updates and client requests.

A server receiving a reference data update hands it off as a PUT or DELETE to Sirius, which runs it through the Paxos protocol and writes it to the persistent transaction

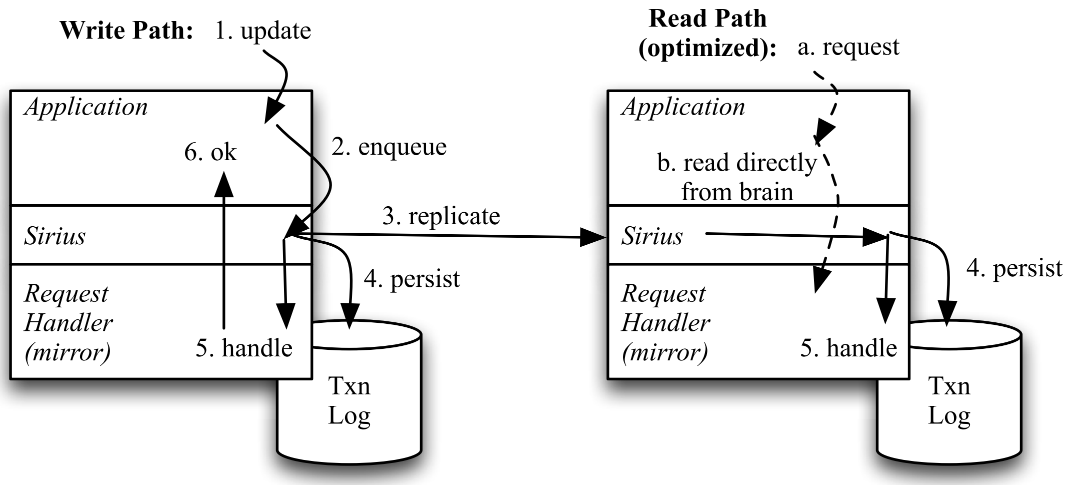


Figure 1: Architecture of a Sirius-based application.

log. The Sirius library on each server then hands the update off to its local mirror to be applied.

The application server can handle read requests in one of two ways. It can hand a request off to its local Sirius, which will serialize it with respect to outstanding updates and then route it as a GET to the mirror. This method would be most appropriate if the application developers wish to have Sirius provide a level of transactional isolation for read requests–ensuring that no updates are applied concurrently to those reads. However, it comes at the expense of serialized access, which may become a performance bottleneck.

In practice, though, our developers use concurrent data structures in the mirror and manage concurrency themselves, bypassing Sirius entirely on the read path.

4 Programming Interface

As we have just seen, a Sirius-based application is conceptually divided into two parts, with the Sirius library as an intermediary. The application proper provides its own interface, for example, exposing HTTP endpoints to receive the reference data publishing events. The application then routes reference data access through Sirius.

After taking care of serialization, replication, and persistence, Sirius invokes a corresponding callback to a *request handler* provided by the application. The request handler takes care of updating or accessing the in-memory representations of the reference data. The application developers are thus completely in control of the native, in-memory representation of this data.

The corresponding programming interfaces are shown in Figure 2; there is a clear correspondence between the Sirius-provided access methods and the application's own request handler. As such, it is easy to imagine a "null Sirius" implementation that would simply invoke the application's request handler directly. This semantic transparency makes it easy to reason functionally about the reference data itself.

```
public interface Sirius {
  Future<SiriusResult>
    enqueueGet(String key);
  Future<SiriusResult>
    enqueuePut(String key, byte[] body);
  Future<SiriusResult>
    enqueueDelete(String key);
  boolean isOnline();
}

public interface RequestHandler {
  SiriusResult handleGet(String key);
  SiriusResult handlePut(String key,
                         byte[] body);
  SiriusResult handleDelete(String key);
}
```

Figure 2: Sirius interfaces. A `SiriusResult` is a Scala case class representing either a captured exception or a successful return, either with or without a return value.

The primary Sirius interface methods are all asynchronous; the contract is that the library itself takes care of scheduling the invocation of the request handlers at the right time to ensure enventual consistency across the cluster nodes. The overall contract is:

- The request handlers for PUTs and DELETEs will be invoked serially and in a consistent order across all nodes.
- Enqueued asynchronous updates will not complete until successful replication has occurred.
- An enqueued GET will be routed locally only, but will be serialized with respect to pending updates.
- At startup time, Sirius will not report itself as "online" until it has completed replay of its transaction log, as indicated by the `isOnline` method.

Sirius does not provide facilities for consistent conditional updates (e.g., compare-and-swap); it merely guarantees consistent ordering of the updates. The low update rate coupled with Sirius not being the system of record mean that the simpler interface presented here has been sufficient in practice.

5 Replication

Updates passed to Sirius via `enqueuePut` or `enqueueDelete` are ordered and replicated via Multi-Paxos with each update being a *command* assigned to a *slot* by the protocol; the slot numbers are recorded as sequence numbers in the persistent log (Section 6). Our implementation fairly faithfully follows the description given by van Renesse [26]. However, there are some slight differences needed to produce a practical implementation and some optimizations made possible by our particular use case.

Stable leader. First, we use the common optimization that disallows continuous "Phase 1" weak leader elections; when a node is pre-empted by a peer with a higher ballot number, it *follows* that peer instead of trying to win the election again. While following, it pings the leader to check for liveness, and forwards any update requests. If the pings fail, a new Phase 1 election begins. This limits vote conflicts, and resultant chattiness, which in turn enhances throughput.

End-to-end retries. Second, because all of the updates are idempotent, we do not track unique request identifiers, as the updates can be retried by the external system of record if a publication attempt is not acknowledged. In turn, this assumption means that we do not need to write the internal process state of the Paxos protocol processes to stable storage to recover from crashes, beyond the persistent log recording decisions assigning updates to specific sequence/slot numbers. This may result in an in-flight update being lost in certain failure scenarios, like a cluster-wide power outage, but as the external client will not have had the write acknowledged, it will eventually time out and retry. This end-to-end design argument for retries allows us to work around the assumption from van Renesse that "a message sent by a non-faulty process to a non-faulty destination process is eventually received (at least once) by the destination process," *i.e.*, that the network is reliable.

Similarly, we bound some processes—notably the "Commander" process that attempts to get a quorum of cluster members to agree on the assignment of an update to a particular slot number—with timeouts and a limited number of retries before giving up on the command. As a practical example, during a long-lived network partition, a minority partition will not be able to make progress, and this prevents the buildup of an unbounded amount of protocol state for attempted but incomplete writes during the partition. In turn, this means Sirius degrades gracefully and does not slow the read path for those nodes, even though their reference datasets in memory may begin to become stale.

Write behind. Finally, the Paxos replicas apply decisions in order by sequence number, buffering any out-of-order decisions as needed. We elected to write the decisions out to the log in sequence number order, as it simplifies log replay and the compaction activities described below in Section 6. We also acknowledge the write once a decision for it has been received, but without waiting for persistence or application to complete; this reduces system write latency and prevents "head-of-line" blocking.

On the other hand, this means that an external publishing client may not get "read your writes" consistency and that there is a window during which an acknowledged write can be lost without having been written to stable storage (*e.g.*, due to a power outage). In practice, neither of these is a problem, as the reference dataset updates are read-only from the point of view of the application's primary customers; similarly, Sirius is not the system of record for the reference dataset, so it is possible to reconstruct lost writes if needed by re-publishing the relevant updates from the upstream system.

State pruning. Once updates have been applied locally, a hint with the maximum sequence number is sent to the local Paxos replica; this allows the in-memory state of proposed updates and recorded decisions to be pruned. We additionally prune proposed but unfinished updates that are older than a given configured cutoff window we expect is longer than an external client's read timeout setting, again relying on the end-to-end retry mechanism to re-publish them.

5.1 Catch-Up Protocol

Because updates must be applied in the same order on all nodes and we also want to log updates to disk in that order, nodes are particularly susceptible to lost decision messages; these delay updates with higher sequence numbers. Therefore, each node periodically selects a random peer and requests a range of updates starting from the lowest sequence number for which it does not have a decision.

The peer replies with all the decisions it has that fall within the given slot number range. Some of these may be returned from a small in-memory cache of updates kept by the peer, especially if the missing decision is a relatively recent one. However, the peer may need to consult the persistent log for older updates no longer in its cache (see Section 6).

If the peer replies with a full range of updates, the process continues: the node requests the next range from the same peer. Once a partial range is returned, the catchup protocol ceases until the next period begins with a new, random peer.

The catchup protocol also provides for dependent cluster members that do not participate in Paxos. Cluster configurations contain only primary members, which thereby know about each other and can participate in Paxos. Other cluster members periodically catch up via the primaries. In practice, this allows a primary "ingest" cluster to feed several dependent application clusters following along for updates—often within seconds of each other and across datacenters, depending upon configuration of the catchup range size and request interval.

In turn, this lets us keep write latencies to a minimum: Paxos only runs across local area networks (LANs). Different clusters can be activated as primaries by pushing a cluster configuration update, which the Sirius library processes without an application restart.

6 Persistence

As updates are ordered by Paxos, Sirius also writes them out to disk in an append-only file. Each record includes an individual record-level checksum, its Paxos sequence number, a timestamp (used for human-readable logging, not for ordering), an operation code (PUT or DELETE), and finally a key and possibly a body (PUTs only), along with related field framing information. This results in variable-sized records, which are not ordinarily a problem: the log is appended by normal write processing, and is normally only read at application startup, where it is scanned sequentially anyway.

There is one exception to this sequential access pattern: while responding to catchup requests, we need to find updates that have fallen out of cache, perhaps because of a node crash or long-lived network partition. In this case, we must find a particular log entry by its sequence number.

At system startup time, Sirius will read the log in order to stream the updates to the application's request handler. At the same time, it will construct a companion index file if one does not already exist. This index file consists of fixed-length records of the form <*seqnum, offset, checksum*>. Although the log file is guaranteed to be sorted in increasing sequence number order, compaction—as we will see shortly—may introduce gaps in the sequence. We perform a binary search on the index file itself to find a particular update number, then use the file offset to locate the update itself in the main log file. Because the index file is small, the operating system's filesystem cache can usually keep it in memory, allowing a quick binary search. When an update is found in the log, any further updates in the requested catchup range can then be streamed without reconsulting the index. In practice, the catchup protocol streams updates to other nodes at least as fast as new updates are written.

6.1 Compaction

Sirius can *compact* its log file: because the PUTs and DELETEs are idempotent, we can remove every log entry for a key except the one with the highest sequence number. Because the overall reference dataset does not grow dramatically in size over time, a compacted log is a relatively compact representation of it; we find that the reference dataset takes up more space in RAM than it does in the log once all the appropriate indices have been created in the application's mirror. This avoids the need for the application to participate in creating snapshots or checkpoints, as in other event-sourced systems [5].

The original Sirius-based application we deployed was under continuous development at the time and was redeployed with new code nearly every day. We took advantage of the application restarts to compact offline during deployments. A separate tool that understands log format would compact the log while a particular application instance was down; the application would then restart by replaying the compacted log. While workable, this was suboptimal: the offline compaction lengthened deployments, and we relied on frequent restarts to prevent the log from getting unwieldy.

Therefore, we developed a scheme for *live compaction* that the Sirius library manages itself in the background. In order to bound the resources (particularly memory) needed by the compaction scheme, we work incrementally by dividing the log into *segments* with a maximum number of entries in each, as in other log-based systems [23, 24]. Each segment is kept in its own directory, and contains a data file and an index file, as described above. Sirius adds new entries only to the highest-numbered segment; when that segment fills up, its file is closed and a new segment is started.

The overall compaction process is as follows: we take a segment s_i and find the set of all keys mentioned in it, $K(s_i)$. Then, for each lowered-numbered segment s_j, we produce a new copy of its data file s'_j that only contains updates to keys not in $K(s_i)$. When s'_j is complete, we swap it into place for s_j atomically. Files are named and managed in a way that allows the compaction process to recover from an application restart; if it finds an existing s'_j file it assumes it is incomplete, removes it, and restarts the compaction of that segment.

After the compaction of the individual segments, we then check if any adjacent segments are small enough to be *merged* and still fit under the maximum segment size; this calculation can be done cheaply by checking the size of the relevant segment index files. The merge operation itself is simply a concatenation of the log and index files for the relevant segments.

Live compaction in Sirius is thus incremental and restartable and does not require a manual operational maintenance step with a separate tool. Thus, while operations staff for a Sirius-based application need to be aware of the segment directory, they do not have to actively manage it. Because all the log segments are regular files, they can be backed up normally, or zipped and copied to another machine. Since the logs are normally append-only, and compaction is incremental, these copies can be taken while the application is running without any special synchronization. We have taken advantage of this to bootstrap new nodes efficiently, especially when seeding a new datacenter, or to copy a production dataset elsewhere for debugging or testing.

7 Experimental Evaluation

In practice we have found Sirius has sufficient write throughput to support our reference data use cases. In this section, we analyze our Sirius implementation experimentally. The library is written in Scala, using the Akka actor library.

All experiments were run on Amazon Web Services (AWS) Elastic Computer Cluster (EC2) servers running a stock 64-bit Linux kernel [2] on "*m1.xlarge*" instances[3] with 4 virtual CPUs and 15 GB RAM. These instances have a 64-bit OpenJDK Java runtime[4] installed; Sirius-based tests use version 1.1.4 of the library.

7.1 Write throughput

Because the optimized read path for an application bypasses Sirius to read directly from the mirror, we are primarily interested in measuring Sirius' write throughput. For these tests, we embed Sirius in a reference web application[5] that exposes a simple key-value store interface via HTTP and uses Java's `ConcurrentHashMap` for its mirror. Load is generated from separate instances running JMeter[6] version 2.11. All requests generate PUTs with 179 byte values (the average object size we see in production use).

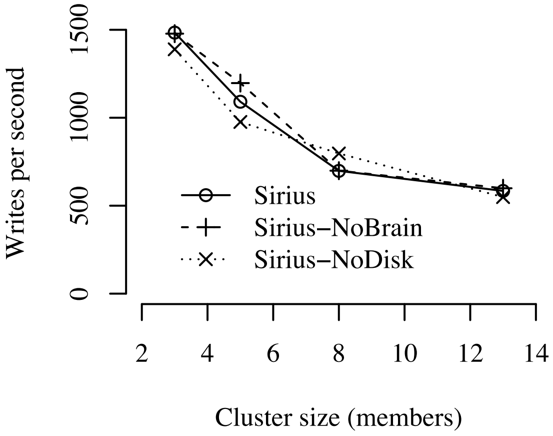


Figure 3: Sirius write throughput.

For these throughput tests, we begin by establishing a baseline under a light load that establishes latency with minimal queueing delay. We then increase load until we find the throughput at which average latency begins to increase; this establishes the maximum practical operating capacity. Our results are summarized in Figure 3.

This experiment shows write throughput for various cluster sizes; it was also repeated for a reference application with a "null" `RequestHandler` (Sirius-NoBrain) and one where disk persistence was turned off (Sirius-NoDisk). There are two main observations to make here:

1. Throughput degrades as cluster size increases. This is primarily due to the quorum-based voting that goes on in Paxos: the more cluster members there are, the more votes are needed for a quorum, and hence the greater chance that there are enough machines sporadically running "slow" (e.g., due to a garbage collection pause) to slow down the algorithm. This trend is consistent with those reported by Hunt *et al.* for ZooKeeper [10].

2. Throughput is not affected by disabling the persistence layer nor by eliminating `RequestHandler` work; we conclude that the Paxos algorithm (or our implementation of it) is the limiting factor.

[2] Kernel `3.4.73-64.112.amzn1.x86_64`.

[3] `http://aws.amazon.com/ec2/instance-types/`

[4] Java version `1.6.0_24`; OpenJDK release `IcedTea6 1.11.14`.

[5] `http://github.com/Comcast/sirius-reference-app`

[6] `http://jmeter.apache.org/`

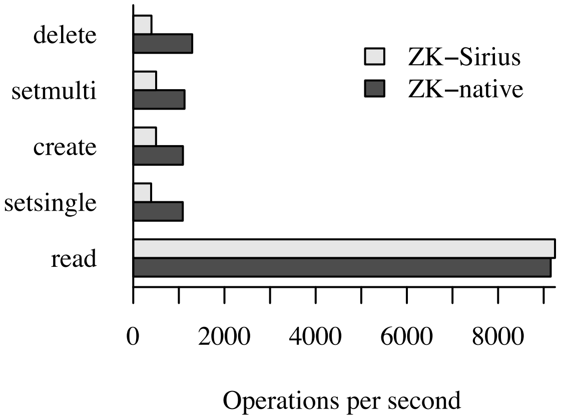


Figure 4: Comparison to ZooKeeper.

7.2 Comparison to ZooKeeper

One of the primary related technologies is ZooKeeper [10]. ZooKeeper offers similar semantics: consistent ordering of writes and eventually consistent reads. However, ZooKeeper also implements a broader set of functionality than Sirius provides, for example, allowing strongly consistent read operations.

We created a modified version of ZooKeeper where the atomic broadcast and persistence layers were replaced with Sirius. This "ZK-Sirius" only supports the core create, read, update, and delete primitives that the systems have in common, but not the full ZooKeeper API. This was sufficient to run benchmarks comparing ZK-Sirius with the native ZooKeeper using a third-party open source benchmarking tool.[7] Because we reused the ZooKeeper interface, we did not have to modify the benchmarking tool at all. The results from testing 5-node clusters are shown in Figure 4. Our prototype and all experiments were based on version 3.4.5 of ZooKeeper.

From the authors' description of the benchmark:

> "The benchmark exercises the ensemble's performance at handling znode reads, repeated writes to a single znode, znode creation, repeated writes to multiple znodes, and znode deletion.... The benchmark connects to each server in the ZooKeeper ensemble using one thread per server."

We conducted the test using the synchronous mode of operation, where each client makes a new request upon receiving the result of the previous one.

As expected, the Sirius-backed version of ZooKeeper achieves essentially identical read performance to the standard ZooKeeper implementation. Throughput on write operations—SETSINGLE, CREATE, SETMULTI and DELETE—is measured to be approximately 40–50% of standard ZooKeeper throughput. Given the read-heavy nature of our reference data workloads, this is a practical tradeoff in order to get the key benefit Sirius provides: namely, in-memory access to the data via arbitrary, native datastructures.

Our point is not to suggest an alternative implementation for ZooKeeper—in particular, Sirius does not support the synchronous reads that are possible with ZooKeeper—but rather to illustrate that our relatively untuned Sirius is in the neighborhood of a highly tuned and production-hardened system like ZooKeeper. This exercise also illustrates two major points:

1. The Sirius programming interface is simple and flexible enough to easily integrate it into ZooKeeper's internals.

2. Distributed system semantics similar to ZooKeeper's—consistently ordered writes with eventually consistent local reads—are available in the form of arbitrary, native data structures.

8 Operational Concerns

In addition to providing a convenient programming interface, we designed Sirius to be operationally friendly. This means that major errors, when they occur, should be obvious and noticeable, but also means that the system should degrade gracefully and preserve as much functionality as possible. Errors and faults are expected, and by and large Sirius manages recovery on its own, although operations staff may intervene if needed. Finally, Sirius has relatively few moving parts that nonetheless provide a lot of operational flexibility.

8.1 Bootstrapping

When a server is first brought online, either as a new cluster member or after recovering from a failure, it may be far behind its active peers. It may have a partial or empty log. While these scenarios do not happen often, they are among the most obvious (and inescapable). With Sirius, there are two fairly straightforward ways to bring such a server back up to date.

First, as mentioned in Section 6, the log is just a collection of regular files that can simply be copied from an existing peer. Second, we can spin up a server with an empty log, and it will use the catch-up protocol to fetch the entire log from a neighbor. Anecdotally, we have bootstrapped several gigabytes of log in minutes this way.

[7] `https://github.com/brownsys/zookeeper-benchmark`

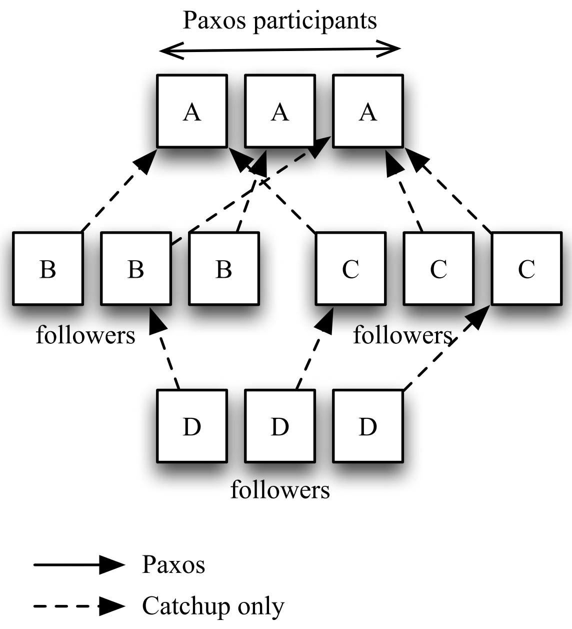


Figure 5: Flexible replication topologies with Sirius.

8.2 Follower Topology

As we described in Section 5, cluster membership is managed with a simple configuration file, with non-Paxos members using the catch-up protocol to "follow" along the primary ingest cluster. This leads to a large amount of topology flexibility; we can, and have, set up small "ingest-only" clusters to process the reference data updates, followed by multiple, larger follower clusters that are scaled out to serve reads. In fact, by specifying different membership files for different clusters, it is possible to follow multiple, redundant clusters and to set up chains of following clusters if desired.

Consider the example topology shown in Figure 5 with four clusters labeled A–D. Clusters A, B, and C share a common configuration that lists the servers of cluster A as primary members; this causes cluster A to participate in Paxos, and clusters B and C to follow the servers in cluster A. Cluster D has a configuration that lists the members of clusters B and C as primaries; this causes cluster D to catch up from randomly selected members of both clusters, even though neither of them are the primary Paxos cluster.

8.3 waltool

To support debugging and operations, we distribute the command-line *waltool* along with Sirius. Waltool primarily allows for manipulation of the Sirius log, and provides the following functionality:

- log format conversion (*e.g.*, from the original unsegmented version to the segmented version and back)
- print the last few entries in the log in human-readable format (similar to the Unix *tail* command)
- log filtering on keys via regular expression—similar to *grep*—to produce a new log that contains or excludes keys matching the expression
- offline compaction
- replay PUT/DELETEs for each update in the log as equivalent HTTP requests sent to a specified server

8.4 Error Handling

Each update in both the index and data files are checksummed. Eventually, a bit will flip, a checksum will fail, and Sirius will refuse to start. Currently, recovery is manual, albeit straightforward: Sirius reports the point at which the problematic record begins. An operator can truncate the log at this point or delete a corrupted index, and Sirius can take care of the rest, rebuilding the index or retrieving the missing updates as needed.

As mentioned above, the liveness of a server's Paxos subsystem is tied to its cluster membership. While rare, we have seen some cases of a cluster failing to make progress. Generally, these resolve themselves (temporary network partitions, DNS flaps), but occasionally they require manual intervention, such as when Paxos has livelocked due to a bug in implementation. We have found that these can almost always be addressed by "power-cycling" Paxos: that is, removing all nodes from the cluster, then putting them back, effectively restarting with a blank slate. This is done without restarting any nodes, simply by changing the monitored configuration file. Nevertheless, situations needing this kind of intervention are rare, and are becoming more rare as we hunt down the last of the bizarre edge cases.

8.5 MBean Server

If supplied with a JMX MBean server[8], Sirius will automatically register many metrics and status markers into it. These include the perceived liveness of the neighboring cluster members, the identity and ballot of the currently elected leader, the number of out-of-order events buffered for persistence, the rolling average time of persisting to disk, the duration of the latest compaction, among others.

[8] See http://en.wikipedia.org/wiki/Java_Management_Extensions.

9 Related Work

> "A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible." –Freeman Dyson

9.1 Consensus

As we described in Section 5, our Paxos [16] implementation follows the one described by van Renesse [26] closely—in fact, our first implementation was a fairly straightforward translation of his pseudocode into Scala/Akka. Similarly, Chandra *et al.* [5] implemented their own state machine description language in C++ in order to get a concise description of the protocol. We also use *master leases* as they describe to optimize throughput of the Multi-Paxos algorithm.

Finally, there are alternative consensus protocols such as RAFT [20], Egalitarian Paxos [19], or ZAB [10] we could have used instead of Paxos without changing the overall application architecture; we will discuss some of these in Section 10. Initially, however, we were attracted to the deep coverage of classic Paxos in the literature to guide our implementation.

9.2 Persistence

Transaction logs are a well-known mechanism for crash recovery in databases, and the Sirius log functions primarily as the *commit log* for Paxos. Unlike more general Paxos implementations like Chubby [5], we do not have to implement application state snapshots or *checkpoints* in Sirius, as the semantics encoded in the log allow us to successfully compact it without assistance from the client application. We also take advantage of the append-only nature of the log in the common-case write path to minimize write latencies to spinning disks, as in journaling file systems [22].

As in log-structured filesystems (LFS) [23], the Sirius log is the primary storage location. We similarly use *segments* as a way to partition the work of compaction, although in our case it is more about enabling faster replay at startup time and bounding the resources needed for continuous compaction than it is about reclaiming free space. Finally, we also create indices mapping sequence numbers to locations within the log, although these are kept separate from the log segments themselves and are essentially hints that can be rebuilt when needed.

Bitcask [24] is a log-structured, persistent hashtable similar in style to the Sirius write-ahead log: it segments the log, periodically merges old segments, and builds *hint files* alongside the segments to provide file addresses for the data bound to particular keys. Indeed, Sirius's log file format is very similar to Bitcask's, although we explicitly differentiate PUTs from DELETEs and also have to record our Paxos sequence numbers. Unlike Bitcask, though, we do not have to build the in-memory *keydir* version of the hint files, as Sirius does not have to provide random read access to the latest update for a particular key. Our index files facilitate, rather, finding updates by sequence number to support catchup.

Finally, LevelDB [1] is a persistent, ordered hashtable that might have handled compaction for us by tracking the most recent updates to particular keys, except that we really have a need for two indices into the set of updates: one by sequence number to support replay and catchup and one by key to support compaction.

9.3 Replicated data structures

There are libraries that provide replicated data structures–typically hashtables–such as Hazelcast [8], Gemfire [21], and Terracotta [25]. While these cover a number of important and practical use cases, they do not permit the use of arbitrary data structures for the replicated data. Although reference datasets are represented as a stream of key-value pairs with Sirius, our applications construct more complex representations in their in-memory mirrors. In particular, the reference data describing television schedules required custom data structures to support all of our use cases efficiently.

9.4 Shared External Logs

Tango [3], like Sirius, provides in-memory *views* that are backed by a shared, persistent log. However, this requires a specialized array of SSD nodes, whereas we needed Sirius to be able to run in a commodity cloud environment. In addition, Tango requires checkpointing support from the client application in order to truncate its logs. Finally, the SSD array, while highly redundant itself, is a single point of failure from the point of view of the client applications, whereas Sirius-related failures are localized to individual cluster nodes.

9.5 External storage

Another option would have been to keep the reference datasets in another system external to the application, such as memcached [6], Redis [17], or ZooKeeper [10]. However, systems like these bring a rich yet ultimately limited set of data structures and require I/O for reads, whether directly by the application programmer or via library calls. In either case, the programmer is still on the hook to provide error handling and proper I/O configuration, something that is difficult for many—if not most—developers to do correctly.

9.6 Event sourcing and message buses

The LMAX architecture [7] builds memory-resident data structures in a single master system while asynchronously replicating the master's transaction log to secondary and tertiary spares. Unlike Sirius, however, this architecture requires the application to participate in producing checkpoints as a way of eventually truncating the log. While the LMAX system posted truly impressive throughput for a single node, ultimately we needed a system that would enable us to scale read operations by adding more servers.

Another approach might have been to use a distributed message bus like Kafka [14] to distribute the updates out to all the cluster nodes. However, Kafka does make some consistency tradeoffs—including the possibility of acknowledged writes being lost [12]—to achieve high throughput. For our reference data use cases, however, we preferred the opposite tradeoff. Kafka also has a limited historical window, which means we would have had to implement checkpointing and replay for our applications anyway.

9.7 In-memory data distribution

Koszewnik [13] describes an entirely different approach to distributing reference data updates, where a single master machine periodically pulls the updates and applies them to an in-memory model. A serialization library, Zeno, then emits optimized deltas of each incremental run to well-known server locations, as well as periodically generating a full checkpoint. This allows downstream clients to individually poll for and apply deltas, and allows new server instances to bootstrap from the most recent checkpoint.

This avoids the need for running a complex consensus algorithm like Paxos, at the expense of having to manually restart the master process if it fails. On the other hand, it dictates the in-memory representation used by all the cluster members, whereas Sirius permits different applications to be part of the same Sirius cluster, sharing the same update stream while building their own customized in-memory representations of the data.

10 Conclusions and Future Work

Overall, we have been happy with Sirius; we will have been using Sirius-powered services in production for almost two years at the time of this paper's publication with few, if any, operational problems. The library's simple and transparent interface, coupled with the ease and control of using native data structures, have led multiple independent teams within Comcast to incorporate Sirius into their services, all to positive effect. Nevertheless, we have identified some opportunities for improvements.

10.1 Paxos Improvements

As with most Multi-Paxos systems, overall write throughput in Sirius is limited by the throughput of the currently-elected leader, and we do experience periodic "bulk load" events where this becomes a bottleneck, albeit a tolerable one to date. Alternative protocols such as Egalitarian Paxos [19] could alleviate this bottleneck with little change to the overall application architecture.

In addition, our cluster configuration is currently statically configured, although our implementation periodically polls its configuration file to watch for updates. Technically, this leaves a window open for inconsistency because cluster membership is not synchronized with the consensus protocol. In practice, however, we are able to pause and buffer the writes into the cluster, switch the configuration, and then resume writing. Consensus protocols like RAFT [20] that integrate cluster membership with consensus could ease our operations.

10.2 Replication

As we described earlier, our WAN replication currently piggybacks on our Paxos catch-up mechanism. Therefore, every member of our downstream non-ingest clusters pulls a copy every update across the WAN. In practice, again, this does not result in a problematic amount of bandwidth, but it is clearly inefficient. Allowing for topology-aware configuration and replication such as those found in Cassandra [15] could allow us to pull fewer (perhaps one) copy of each update across the WAN, before then further replicating locally.

10.3 Replay

In practice, since our reference datasets fit in memory, so do their representations in our write-ahead logs. This means the system read-ahead caches do a good job at the I/O required for the linear scans necessary for replay at system startup time. Still, there is a bottleneck where Sirius passes the updates synchronously and serially to the application's `RequestHandler`; an alternative mechanism for safely processing some updates in parallel would be desirable.

10.4 Conclusions

In this paper, we have described a novel architectural approach for handling application reference data centered around a new distributed system library, Sirius. A Sirius-based architecture allows for:

- in-memory, local access to the reference data in arbitrary data structures;
- eventually consistent replication of updates;
- local persistence and replay of updates;
- with a semantically-transparent library interface.

The Sirius library is available under the Apache 2 License from: `http://github.com/Comcast/sirius`.

Acknowledgments

We would like to thank the anonymous reviewers for their comments; the paper is clearer and more concise for your efforts. We would additionally like to thank the CIM Platform/API team at Comcast, Sirius' first users and development collaborators; Sirius would not have been possible without your help and hard work.

References

[1] Leveldb website. `https://code.google.com/p/leveldb/`. [Online; accessed 16-Jan-2014].

[2] Amazon Web Services, Inc. Daily Global Weather Measurements, 1929-2009 (NCDC, GSOD). `http://aws.amazon.com/datasets/2759`, 2013. [Online; accessed 8-January-2014].

[3] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 325–340.

[4] Board of Governors of the Federal Reserve System. FRB H10: Data download program. `http://www.federalreserve.gov/datadownload/Choose.aspx?rel=H10`, January 2014. [Online; accessed 8-January-2014].

[5] Chandra, T. D., Griesemer, R., and Redstone, J. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[6] Fitzpatrick, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (Aug. 2004).

[7] Fowler, M. The LMAX architecture. `http://martinfowler.com/articles/lmax.html`, July 2011. [Online; accessed 17-Jan-2014].

[8] Hazelcast, Inc. `http://hazelcast.org`, 2014. [Online; accessed 23-January-2014].

[9] Hoff, T. Latency is everywhere and it costs you sales: How to crush it. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`, July 2009. *High Scalability*, blog. [Online; accessed 20-January-2014].

[10] 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), USENIX-ATC'10, USENIX Association, pp. 11–11.

[11] IMDb.com, Inc. Advanced title search. `http://www.imdb.com/search/title`. [Online; accessed 24-January-2014].

[12] Kingsbury, K. Call me maybe: Kakfa. `http://aphyr.com/posts/293-call-me-maybe-kafka`, September 2013. [Online; accessed 17-Jan-2014].

[13] Koszewnik, D. Announcing Zeno—Netflix's in-memory data distribution framework. `http://techblog.netflix.com/2013/12/announcing-zeno-netflixs-in-memory-data.html`, December 2013. Netflix Tech Blog. [Online; accessed 20-January-2014].

[14] Kreps, J., Narkhede, N., and Rao, J. Kafka: a distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases* (New York, NY, USA, June 2011), NetDB 2011, ACM.

[15] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[16] Lamport, L. The part-time parliament. *ACM Transactions on Computing Systems (TOCS) 16*, 2 (May 1998), 133–169.

[17] Lerner, R. M. At the forge: Redis. *Linux Journal 2010*, 197 (Sept. 2010).

[18] Miller, A. Hibernate query cache considered harmful? `http://tech.puredanger.com/2009/07/10/hibernate-query-cache/`, July 2009. *Pure Danger Tech*, blog. [Online; accessed 8-January-2014].

[19] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358–372.

[20] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. `https://ramcloud.stanford.edu/raft.pdf`, October 2013. [Online; accessed 13-Jan-2014].

[21] Pivotal Software, Inc. Pivotal gemfire. `http://gopivotal.com/products/pivotal-gemfire`. [Online; accessed 16-Jan-2014].

[22] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), ATEC '05, USENIX Association, pp. 8–8.

[23] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

[24] Sheehy, J., and Smith, D. Bitcask: A log-structured hash table for fast key/value data. `http://downloads.basho.com/papers/bitcask-intro.pdf`. [Online; accessed 16-Jan-2014].

[25] Terracotta, Inc. *The Definitive Guide to Terracotta: Cluster the JVM for Spring, Hibernate and POJO Scalability*. Apress, Berkeley, CA, USA, 2008.

[26] van Renesse, R. Paxos made moderately complex. `http://www.cs.cornell.edu/courses/cs7412/2011sp/paxos.pdf`, March 2011. [Online; accessed 13-Jan-2014].

[27] Wikipedia. Encyclopædia Britannica Ultimate Reference Suite — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/wiki/Encyclop%C3%A6dia_Britannica_Ultimate_Reference_Suite`, 2013. [Online; accessed 8-January-2014].

In Search of an Understandable Consensus Algorithm

Diego Ongaro and John Ousterhout
Stanford University

Abstract

Raft is a consensus algorithm for managing a replicated log. It produces a result equivalent to (multi-)Paxos, and it is as efficient as Paxos, but its structure is different from Paxos; this makes Raft more understandable than Paxos and also provides a better foundation for building practical systems. In order to enhance understandability, Raft separates the key elements of consensus, such as leader election, log replication, and safety, and it enforces a stronger degree of coherency to reduce the number of states that must be considered. Results from a user study demonstrate that Raft is easier for students to learn than Paxos. Raft also includes a new mechanism for changing the cluster membership, which uses overlapping majorities to guarantee safety.

1 Introduction

Consensus algorithms allow a collection of machines to work as a coherent group that can survive the failures of some of its members. Because of this, they play a key role in building reliable large-scale software systems. Paxos [13, 14] has dominated the discussion of consensus algorithms over the last decade: most implementations of consensus are based on Paxos or influenced by it, and Paxos has become the primary vehicle used to teach students about consensus.

Unfortunately, Paxos is quite difficult to understand, in spite of numerous attempts to make it more approachable. Furthermore, its architecture requires complex changes to support practical systems. As a result, both system builders and students struggle with Paxos.

After struggling with Paxos ourselves, we set out to find a new consensus algorithm that could provide a better foundation for system building and education. Our approach was unusual in that our primary goal was *understandability*: could we define a consensus algorithm for practical systems and describe it in a way that is significantly easier to learn than Paxos? Furthermore, we wanted the algorithm to facilitate the development of intuitions that are essential for system builders. It was important not just for the algorithm to work, but for it to be obvious why it works.

The result of this work is a consensus algorithm called Raft. In designing Raft we applied specific techniques to improve understandability, including decomposition (Raft separates leader election, log replication, and safety) and state space reduction (relative to Paxos, Raft reduces the degree of nondeterminism and the ways servers can be inconsistent with each other). A user study with 43 students at two universities shows that Raft is significantly easier to understand than Paxos: after learning both algorithms, 33 of these students were able to answer questions about Raft better than questions about Paxos.

Raft is similar in many ways to existing consensus algorithms (most notably, Oki and Liskov's Viewstamped Replication [27, 20]), but it has several novel features:

- **Strong leader:** Raft uses a stronger form of leadership than other consensus algorithms. For example, log entries only flow from the leader to other servers. This simplifies the management of the replicated log and makes Raft easier to understand.
- **Leader election:** Raft uses randomized timers to elect leaders. This adds only a small amount of mechanism to the heartbeats already required for any consensus algorithm, while resolving conflicts simply and rapidly.
- **Membership changes:** Raft's mechanism for changing the set of servers in the cluster uses a new *joint consensus* approach where the majorities of two different configurations overlap during transitions. This allows the cluster to continue operating normally during configuration changes.

We believe that Raft is superior to Paxos and other consensus algorithms, both for educational purposes and as a foundation for implementation. It is simpler and more understandable than other algorithms; it is described completely enough to meet the needs of a practical system; it has several open-source implementations and is used by several companies; its safety properties have been formally specified and proven; and its efficiency is comparable to other algorithms.

The remainder of the paper introduces the replicated state machine problem (Section 2), discusses the strengths and weaknesses of Paxos (Section 3), describes our general approach to understandability (Section 4), presents the Raft consensus algorithm (Sections 5–7), evaluates Raft (Section 8), and discusses related work (Section 9). A few elements of the Raft algorithm have been omitted here because of space limitations, but they are available in an extended technical report [29]. The additional material describes how clients interact with the system, and how space in the Raft log can be reclaimed.

2 Replicated state machines

Consensus algorithms typically arise in the context of *replicated state machines* [33]. In this approach, state machines on a collection of servers compute identical copies of the same state and can continue operating even if some of the servers are down. Replicated state machines are used to solve a variety of fault tolerance problems in distributed systems. For example, large-scale systems that

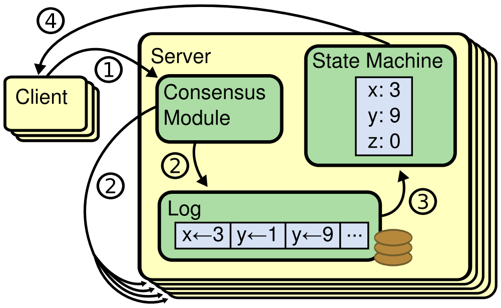


Figure 1: Replicated state machine architecture. The consensus algorithm manages a replicated log containing state machine commands from clients. The state machines process identical sequences of commands from the logs, so they produce the same outputs.

have a single cluster leader, such as GFS [7], HDFS [34], and RAMCloud [30], typically use a separate replicated state machine to manage leader election and store configuration information that must survive leader crashes. Examples of replicated state machines include Chubby [2] and ZooKeeper [9].

Replicated state machines are typically implemented using a replicated log, as shown in Figure 1. Each server stores a log containing a series of commands, which its state machine executes in order. Each log contains the same commands in the same order, so each state machine processes the same sequence of commands. Since the state machines are deterministic, each computes the same state and the same sequence of outputs.

Keeping the replicated log consistent is the job of the consensus algorithm. The consensus module on a server receives commands from clients and adds them to its log. It communicates with the consensus modules on other servers to ensure that every log eventually contains the same requests in the same order, even if some servers fail. Once commands are properly replicated, each server's state machine processes them in log order, and the outputs are returned to clients. As a result, the servers appear to form a single, highly reliable state machine.

Consensus algorithms for practical systems typically have the following properties:

- They ensure *safety* (never returning an incorrect result) under all non-Byzantine conditions, including network delays, partitions, and packet loss, duplication, and reordering.
- They are fully functional (*available*) as long as any majority of the servers are operational and can communicate with each other and with clients. Thus, a typical cluster of five servers can tolerate the failure of any two servers. Servers are assumed to fail by stopping; they may later recover from state on stable storage and rejoin the cluster.
- They do not depend on timing to ensure the consistency of the logs: faulty clocks and extreme message delays can, at worst, cause availability problems.
- In the common case, a command can complete as soon as a majority of the cluster has responded to a single round of remote procedure calls; a minority of slow servers need not impact overall system performance.

3 What's wrong with Paxos?

Over the last ten years, Leslie Lamport's Paxos protocol [13] has become almost synonymous with consensus: it is the protocol most commonly taught in courses, and most implementations of consensus use it as a starting point. Paxos first defines a protocol capable of reaching agreement on a single decision, such as a single replicated log entry. We refer to this subset as *single-decree Paxos*. Paxos then combines multiple instances of this protocol to facilitate a series of decisions such as a log (*multi-Paxos*). Paxos ensures both safety and liveness, and it supports changes in cluster membership. Its correctness has been proven, and it is efficient in the normal case.

Unfortunately, Paxos has two significant drawbacks. The first drawback is that Paxos is exceptionally difficult to understand. The full explanation [13] is notoriously opaque; few people succeed in understanding it, and only with great effort. As a result, there have been several attempts to explain Paxos in simpler terms [14, 18, 19]. These explanations focus on the single-decree subset, yet they are still challenging. In an informal survey of attendees at NSDI 2012, we found few people who were comfortable with Paxos, even among seasoned researchers. We struggled with Paxos ourselves; we were not able to understand the complete protocol until after reading several simplified explanations and designing our own alternative protocol, a process that took almost a year.

We hypothesize that Paxos' opaqueness derives from its choice of the single-decree subset as its foundation. Single-decree Paxos is dense and subtle: it is divided into two stages that do not have simple intuitive explanations and cannot be understood independently. Because of this, it is difficult to develop intuitions about why the single-decree protocol works. The composition rules for multi-Paxos add significant additional complexity and subtlety. We believe that the overall problem of reaching consensus on multiple decisions (i.e., a log instead of a single entry) can be decomposed in other ways that are more direct and obvious.

The second problem with Paxos is that it does not provide a good foundation for building practical implementations. One reason is that there is no widely agreed-upon algorithm for multi-Paxos. Lamport's descriptions are mostly about single-decree Paxos; he sketched possible approaches to multi-Paxos, but many details are missing. There have been several attempts to flesh out and optimize Paxos, such as [24], [35], and [11], but these differ from each other and from Lamport's sketches. Systems such as Chubby [4] have implemented Paxos-like algorithms, but in most cases their details have not been published.

Furthermore, the Paxos architecture is a poor one for building practical systems; this is another consequence of the single-decree decomposition. For example, there is little benefit to choosing a collection of log entries independently and then melding them into a sequential log; this just adds complexity. It is simpler and more efficient to design a system around a log, where new entries are appended sequentially in a constrained order. Another problem is that Paxos uses a symmetric peer-to-peer approach at its core (though it eventually suggests a weak form of leadership as a performance optimization). This makes sense in a simplified world where only one decision will be made, but few practical systems use this approach. If a series of decisions must be made, it is simpler and faster to first elect a leader, then have the leader coordinate the decisions.

As a result, practical systems bear little resemblance to Paxos. Each implementation begins with Paxos, discovers the difficulties in implementing it, and then develops a significantly different architecture. This is time-consuming and error-prone, and the difficulties of understanding Paxos exacerbate the problem. Paxos' formulation may be a good one for proving theorems about its correctness, but real implementations are so different from Paxos that the proofs have little value. The following comment from the Chubby implementers is typical:

> There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system.... the final system will be based on an unproven protocol [4].

Because of these problems, we concluded that Paxos does not provide a good foundation either for system building or for education. Given the importance of consensus in large-scale software systems, we decided to see if we could design an alternative consensus algorithm with better properties than Paxos. Raft is the result of that experiment.

4 Designing for understandability

We had several goals in designing Raft: it must provide a complete and practical foundation for system building, so that it significantly reduces the amount of design work required of developers; it must be safe under all conditions and available under typical operating conditions; and it must be efficient for common operations. But our most important goal—and most difficult challenge—was *understandability*. It must be possible for a large audience to understand the algorithm comfortably. In addition, it must be possible to develop intuitions about the algorithm, so that system builders can make the extensions that are inevitable in real-world implementations.

There were numerous points in the design of Raft where we had to choose among alternative approaches. In these situations we evaluated the alternatives based on understandability: how hard is it to explain each alternative (for example, how complex is its state space, and does it have subtle implications?), and how easy will it be for a reader to completely understand the approach and its implications?

We recognize that there is a high degree of subjectivity in such analysis; nonetheless, we used two techniques that are generally applicable. The first technique is the well-known approach of problem decomposition: wherever possible, we divided problems into separate pieces that could be solved, explained, and understood relatively independently. For example, in Raft we separated leader election, log replication, safety, and membership changes.

Our second approach was to simplify the state space by reducing the number of states to consider, making the system more coherent and eliminating nondeterminism where possible. Specifically, logs are not allowed to have holes, and Raft limits the ways in which logs can become inconsistent with each other. Although in most cases we tried to eliminate nondeterminism, there are some situations where nondeterminism actually improves understandability. In particular, randomized approaches introduce nondeterminism, but they tend to reduce the state space by handling all possible choices in a similar fashion ("choose any; it doesn't matter"). We used randomization to simplify the Raft leader election algorithm.

5 The Raft consensus algorithm

Raft is an algorithm for managing a replicated log of the form described in Section 2. Figure 2 summarizes the algorithm in condensed form for reference, and Figure 3 lists key properties of the algorithm; the elements of these figures are discussed piecewise over the rest of this section.

Raft implements consensus by first electing a distinguished *leader*, then giving the leader complete responsibility for managing the replicated log. The leader accepts log entries from clients, replicates them on other servers, and tells servers when it is safe to apply log entries to their state machines. Having a leader simplifies the management of the replicated log. For example, the leader can decide where to place new entries in the log without consulting other servers, and data flows in a simple fashion from the leader to other servers. A leader can fail or become disconnected from the other servers, in which case a new leader is elected.

Given the leader approach, Raft decomposes the consensus problem into three relatively independent subproblems, which are discussed in the subsections that follow:

- **Leader election:** a new leader must be chosen when an existing leader fails (Section 5.2).
- **Log replication:** the leader must accept log entries from clients and replicate them across the cluster, forcing the other logs to agree with its own (Section 5.3).
- **Safety:** the key safety property for Raft is the State Ma-

State

Persistent state on all servers:

(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:

(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [28] describes the algorithm more precisely.

Election Safety: at most one leader can be elected in a given term. §5.2

Leader Append-Only: a leader never overwrites or deletes entries in its log; it only appends new entries. §5.3

Log Matching: if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index. §5.3

Leader Completeness: if a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms. §5.4

State Machine Safety: if a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index. §5.4.3

Figure 3: Raft guarantees that each of these properties is true at all times. The section numbers indicate where each property is discussed.

chine Safety Property in Figure 3: if any server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log index. Section 5.4 describes how Raft ensures this property; the solution involves an additional restriction on the election mechanism described in Section 5.2.

After presenting the consensus algorithm, this section discusses the issue of availability and the role of timing in the system.

5.1 Raft basics

A Raft cluster contains several servers; five is a typical number, which allows the system to tolerate two failures. At any given time each server is in one of three states: *leader*, *follower*, or *candidate*. In normal operation there is exactly one leader and all of the other servers are followers. Followers are passive: they issue no requests on their own but simply respond to requests from leaders and candidates. The leader handles all client requests (if a client contacts a follower, the follower redirects it to the leader). The third state, candidate, is used to elect a new leader as described in Section 5.2. Figure 4 shows the states and their transitions; the transitions are discussed below.

Raft divides time into *terms* of arbitrary length, as shown in Figure 5. Terms are numbered with consecutive integers. Each term begins with an *election*, in which one or more candidates attempt to become leader as described in Section 5.2. If a candidate wins the election, then it serves as leader for the rest of the term. In some situations an election will result in a split vote. In this case the term will end with no leader; a new term (with a new election) will begin shortly. Raft ensures that there is at most one leader in a given term.

Different servers may observe the transitions between terms at different times, and in some situations a server may not observe an election or even entire terms. Terms act as a logical clock [12] in Raft, and they allow servers to detect obsolete information such as stale leaders. Each server stores a *current term* number, which increases monotonically over time. Current terms are exchanged whenever servers communicate; if one server's current term is smaller than the other's, then it updates its current term to the larger value. If a candidate or leader discovers that its term is out of date, it immediately reverts to follower state. If a server receives a request with a stale term number, it rejects the request.

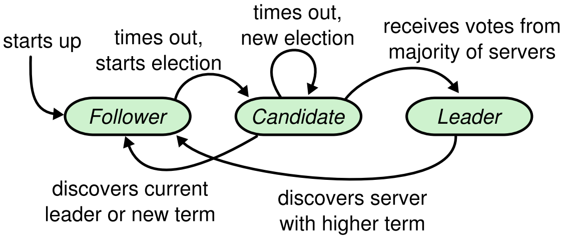

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

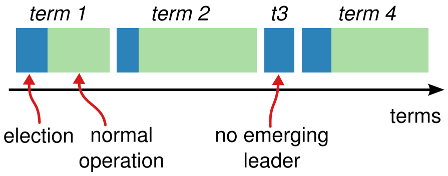


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

Raft servers communicate using remote procedure calls (RPCs), and the consensus algorithm requires only two types of RPCs. RequestVote RPCs are initiated by candidates during elections (Section 5.2), and AppendEntries RPCs are initiated by leaders to replicate log entries and to provide a form of heartbeat (Section 5.3). Servers retry RPCs if they do not receive a response in a timely manner, and they issue RPCs in parallel for best performance.

5.2 Leader election

Raft uses a heartbeat mechanism to trigger leader election. When servers start up, they begin as followers. A server remains in follower state as long as it receives valid RPCs from a leader or candidate. Leaders send periodic heartbeats (AppendEntries RPCs that carry no log entries) to all followers in order to maintain their authority. If a follower receives no communication over a period of time called the *election timeout*, then it assumes there is no viable leader and begins an election to choose a new leader.

To begin an election, a follower increments its current term and transitions to candidate state. It then votes for itself and issues RequestVote RPCs in parallel to each of the other servers in the cluster. A candidate continues in this state until one of three things happens: (a) it wins the election, (b) another server establishes itself as leader, or (c) a period of time goes by with no winner. These outcomes are discussed separately in the paragraphs below.

A candidate wins an election if it receives votes from a majority of the servers in the full cluster for the same term. Each server will vote for at most one candidate in a given term, on a first-come-first-served basis (note: Section 5.4 adds an additional restriction on votes). The majority rule ensures that at most one candidate can win the election for a particular term (the Election Safety Property in Figure 3). Once a candidate wins an election, it becomes leader. It then sends heartbeat messages to all of the other servers to establish its authority and prevent new elections.

While waiting for votes, a candidate may receive an AppendEntries RPC from another server claiming to be leader. If the leader's term (included in its RPC) is at least as large as the candidate's current term, then the candidate recognizes the leader as legitimate and returns to follower state. If the term in the RPC is smaller than the candidate's current term, then the candidate rejects the RPC and continues in candidate state.

The third possible outcome is that a candidate neither wins nor loses the election: if many followers become candidates at the same time, votes could be split so that no candidate obtains a majority. When this happens, each candidate will time out and start a new election by incrementing its term and initiating another round of RequestVote RPCs. However, without extra measures split votes could repeat indefinitely.

Raft uses randomized election timeouts to ensure that split votes are rare and that they are resolved quickly. To prevent split votes in the first place, election timeouts are chosen randomly from a fixed interval (e.g., 150–300ms). This spreads out the servers so that in most cases only a single server will time out; it wins the election and sends heartbeats before any other servers time out. The same mechanism is used to handle split votes. Each candidate restarts its randomized election timeout at the start of an election, and it waits for that timeout to elapse before starting the next election; this reduces the likelihood of another split vote in the new election. Section 8.3 shows that this approach elects a leader rapidly.

Elections are an example of how understandability guided our choice between design alternatives. Initially we planned to use a ranking system: each candidate was assigned a unique rank, which was used to select between competing candidates. If a candidate discovered another candidate with higher rank, it would return to follower state so that the higher ranking candidate could more easily win the next election. We found that this approach created subtle issues around availability (a lower-ranked server might need to time out and become a candidate again if a higher-ranked server fails, but if it does so too soon, it can reset progress towards electing a leader). We made adjustments to the algorithm several times, but after each adjustment new corner cases appeared. Eventually we concluded that the randomized retry approach is more obvious and understandable.

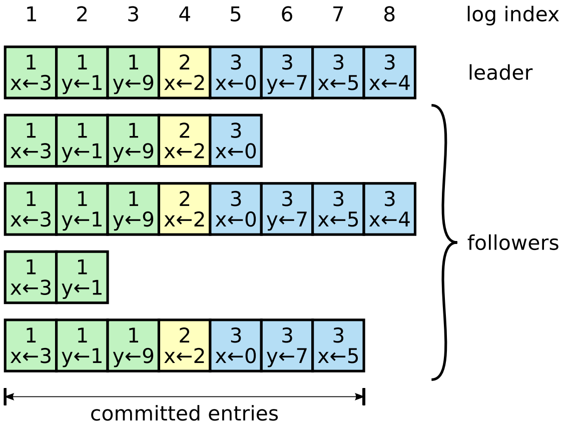

1 2 3 4 5 6 7 8
log index
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
leader
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5 | 3 x←4
1 x←3 | 1 y←1
1 x←3 | 1 y←1 | 1 y←9 | 2 x←2 | 3 x←0 | 3 y←7 | 3 x←5
followers
committed entries

Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

5.3 Log replication

Once a leader has been elected, it begins servicing client requests. Each client request contains a command to be executed by the replicated state machines. The leader appends the command to its log as a new entry, then issues AppendEntries RPCs in parallel to each of the other servers to replicate the entry. When the entry has been safely replicated (as described below), the leader applies the entry to its state machine and returns the result of that execution to the client. If followers crash or run slowly, or if network packets are lost, the leader retries AppendEntries RPCs indefinitely (even after it has responded to the client) until all followers eventually store all log entries.

Logs are organized as shown in Figure 6. Each log entry stores a state machine command along with the term number when the entry was received by the leader. The term numbers in log entries are used to detect inconsistencies between logs and to ensure some of the properties in Figure 3. Each log entry also has an integer index identifying its position in the log.

The leader decides when it is safe to apply a log entry to the state machines; such an entry is called *committed*. Raft guarantees that committed entries are durable and will eventually be executed by all of the available state machines. A log entry is committed once the leader that created the entry has replicated it on a majority of the servers (e.g., entry 7 in Figure 6). This also commits all preceding entries in the leader's log, including entries created by previous leaders. Section 5.4 discusses some subtleties when applying this rule after leader changes, and it also shows that this definition of commitment is safe. The leader keeps track of the highest index it knows to be committed, and it includes that index in future AppendEntries RPCs (including heartbeats) so that the other servers eventually find out. Once a follower learns that a log entry is committed, it applies the entry to its local state machine (in log order).

We designed the Raft log mechanism to maintain a high level of coherency between the logs on different servers. Not only does this simplify the system's behavior and make it more predictable, but it is an important component of ensuring safety. Raft maintains the following properties, which together constitute the Log Matching Property in Figure 3:

- If two entries in different logs have the same index and term, then they store the same command.
- If two entries in different logs have the same index and term, then the logs are identical in all preceding entries.

The first property follows from the fact that a leader creates at most one entry with a given log index in a given term, and log entries never change their position in the log. The second property is guaranteed by a simple consistency check performed by AppendEntries. When sending an AppendEntries RPC, the leader includes the index and term of the entry in its log that immediately precedes the new entries. If the follower does not find an entry in its log with the same index and term, then it refuses the new entries. The consistency check acts as an induction step: the initial empty state of the logs satisfies the Log Matching Property, and the consistency check preserves the Log Matching Property whenever logs are extended. As a result, whenever AppendEntries returns successfully, the leader knows that the follower's log is identical to its own log up through the new entries.

During normal operation, the logs of the leader and followers stay consistent, so the AppendEntries consistency check never fails. However, leader crashes can leave the logs inconsistent (the old leader may not have fully replicated all of the entries in its log). These inconsistencies can compound over a series of leader and follower crashes. Figure 7 illustrates the ways in which followers' logs may differ from that of a new leader. A follower may be missing entries that are present on the leader, it may have extra entries that are not present on the leader, or both. Missing and extraneous entries in a log may span multiple terms.

In Raft, the leader handles inconsistencies by forcing the followers' logs to duplicate its own. This means that conflicting entries in follower logs will be overwritten with entries from the leader's log. Section 5.4 will show that this is safe when coupled with one more restriction.

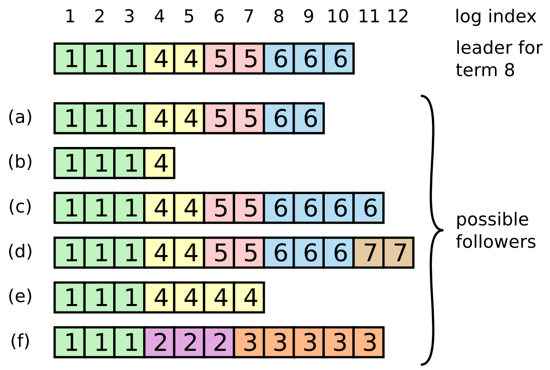


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

To bring a follower's log into consistency with its own, the leader must find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and send the follower all of the leader's entries after that point. All of these actions happen in response to the consistency check performed by AppendEntries RPCs. The leader maintains a *nextIndex* for each follower, which is the index of the next log entry the leader will send to that follower. When a leader first comes to power, it initializes all nextIndex values to the index just after the last one in its log (11 in Figure 7). If a follower's log is inconsistent with the leader's, the AppendEntries consistency check will fail in the next AppendEntries RPC. After a rejection, the leader decrements nextIndex and retries the AppendEntries RPC. Eventually nextIndex will reach a point where the leader and follower logs match. When this happens, AppendEntries will succeed, which removes any conflicting entries in the follower's log and appends entries from the leader's log (if any). Once AppendEntries succeeds, the follower's log is consistent with the leader's, and it will remain that way for the rest of the term.

The protocol can be optimized to reduce the number of rejected AppendEntries RPCs; see [29] for details.

With this mechanism, a leader does not need to take any special actions to restore log consistency when it comes to power. It just begins normal operation, and the logs automatically converge in response to failures of the AppendEntries consistency check. A leader never overwrites or deletes entries in its own log (the Leader Append-Only Property in Figure 3).

This log replication mechanism exhibits the desirable consensus properties described in Section 2: Raft can accept, replicate, and apply new log entries as long as a majority of the servers are up; in the normal case a new entry can be replicated with a single round of RPCs to a majority of the cluster; and a single slow follower will not impact performance.

5.4 Safety

The previous sections described how Raft elects leaders and replicates log entries. However, the mechanisms described so far are not quite sufficient to ensure that each state machine executes exactly the same commands in the same order. For example, a follower might be unavailable while the leader commits several log entries, then it could be elected leader and overwrite these entries with new ones; as a result, different state machines might execute different command sequences.

This section completes the Raft algorithm by adding a restriction on which servers may be elected leader. The restriction ensures that the leader for any given term contains all of the entries committed in previous terms (the Leader Completeness Property from Figure 3). Given the election restriction, we then make the rules for commitment more precise. Finally, we present a proof sketch for the Leader Completeness Property and show how it leads to correct behavior of the replicated state machine.

5.4.1 Election restriction

In any leader-based consensus algorithm, the leader must eventually store all of the committed log entries. In some consensus algorithms, such as Viewstamped Replication [20], a leader can be elected even if it doesn't initially contain all of the committed entries. These algorithms contain additional mechanisms to identify the missing entries and transmit them to the new leader, either during the election process or shortly afterwards. Unfortunately, this results in considerable additional mechanism and complexity. Raft uses a simpler approach where it guarantees that all the committed entries from previous terms are present on each new leader from the moment of its election, without the need to transfer those entries to the leader. This means that log entries only flow in one direction, from leaders to followers, and leaders never overwrite existing entries in their logs.

Raft uses the voting process to prevent a candidate from winning an election unless its log contains all committed entries. A candidate must contact a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log in that majority (where "up-to-date" is defined precisely below), then it will hold all the committed entries. The RequestVote RPC implements this restriction: the RPC includes information about the candidate's log, and the voter denies its vote if its own log is more up-to-date than that of the candidate.

Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. If the logs have last entries with different terms, then the log with the later term is more up-to-date. If the logs end with the same term, then whichever log is longer is more up-to-date.

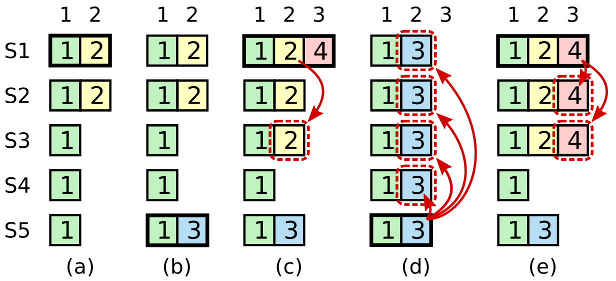


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

5.4.2 Committing entries from previous terms

As described in Section 5.3, a leader knows that an entry from its current term is committed once that entry is stored on a majority of the servers. If a leader crashes before committing an entry, future leaders will attempt to finish replicating the entry. However, a leader cannot immediately conclude that an entry from a previous term is committed once it is stored on a majority of servers. Figure 8 illustrates a situation where an old log entry is stored on a majority of servers, yet can still be overwritten by a future leader.

To eliminate problems like the one in Figure 8, Raft never commits log entries from previous terms by counting replicas. Only log entries from the leader's current term are committed by counting replicas; once an entry from the current term has been committed in this way, then all prior entries are committed indirectly because of the Log Matching Property. There are some situations where a leader could safely conclude that an older log entry is committed (for example, if that entry is stored on every server), but Raft takes a more conservative approach for simplicity.

Raft incurs this extra complexity in the commitment rules because log entries retain their original term numbers when a leader replicates entries from previous terms. In other consensus algorithms, if a new leader rereplicates entries from prior "terms," it must do so with its new "term number." Raft's approach makes it easier

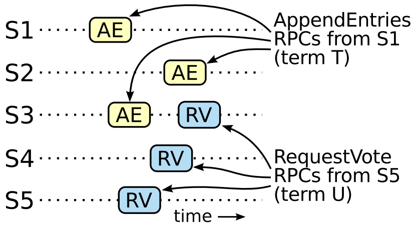

S1
S2
S3
S4
S5
AE
AE
AE
RV
RV
RV
AppendEntries RPCs from S1 (term T)
RequestVote RPCs from S5 (term U)
time →

Figure 9: If S1 (leader for term T) commits a new log entry from its term, and S5 is elected leader for a later term U, then there must be at least one server (S3) that accepted the log entry and also voted for S5.

to reason about log entries, since they maintain the same term number over time and across logs. In addition, new leaders in Raft send fewer log entries from previous terms than in other algorithms (other algorithms must send redundant log entries to renumber them before they can be committed).

5.4.3 Safety argument

Given the complete Raft algorithm, we can now argue more precisely that the Leader Completeness Property holds (this argument is based on the safety proof; see Section 8.2). We assume that the Leader Completeness Property does not hold, then we prove a contradiction. Suppose the leader for term T (leader$_T$) commits a log entry from its term, but that log entry is not stored by the leader of some future term. Consider the smallest term U $>$ T whose leader (leader$_U$) does not store the entry.

1. The committed entry must have been absent from leader$_U$'s log at the time of its election (leaders never delete or overwrite entries).

2. leader$_T$ replicated the entry on a majority of the cluster, and leader$_U$ received votes from a majority of the cluster. Thus, at least one server ("the voter") both accepted the entry from leader$_T$ and voted for leader$_U$, as shown in Figure 9. The voter is key to reaching a contradiction.

3. The voter must have accepted the committed entry from leader$_T$ *before* voting for leader$_U$; otherwise it would have rejected the AppendEntries request from leader$_T$ (its current term would have been higher than T).

4. The voter still stored the entry when it voted for leader$_U$, since every intervening leader contained the entry (by assumption), leaders never remove entries, and followers only remove entries if they conflict with the leader.

5. The voter granted its vote to leader$_U$, so leader$_U$'s log must have been as up-to-date as the voter's. This leads to one of two contradictions.

6. First, if the voter and leader$_U$ shared the same last log term, then leader$_U$'s log must have been at least as long as the voter's, so its log contained every entry in the voter's log. This is a contradiction, since the voter contained the committed entry and leader$_U$ was assumed not to.

7. Otherwise, leader$_U$'s last log term must have been larger than the voter's. Moreover, it was larger than T, since the voter's last log term was at least T (it contains the committed entry from term T). The earlier leader that created leader$_U$'s last log entry must have contained the committed entry in its log (by assumption). Then, by the Log Matching Property, leader$_U$'s log must also contain the committed entry, which is a contradiction.

8. This completes the contradiction. Thus, the leaders of all terms greater than T must contain all entries from term T that are committed in term T.

9. The Log Matching Property guarantees that future leaders will also contain entries that are committed indirectly, such as index 2 in Figure 8(d).

Given the Leader Completeness Property, it is easy to prove the State Machine Safety Property from Figure 3 and that all state machines apply the same log entries in the same order (see [29]).

5.5 Follower and candidate crashes

Until this point we have focused on leader failures. Follower and candidate crashes are much simpler to handle than leader crashes, and they are both handled in the same way. If a follower or candidate crashes, then future RequestVote and AppendEntries RPCs sent to it will fail. Raft handles these failures by retrying indefinitely; if the crashed server restarts, then the RPC will complete successfully. If a server crashes after completing an RPC but before responding, then it will receive the same RPC again after it restarts. Raft RPCs are idempotent, so this causes no harm. For example, if a follower receives an AppendEntries request that includes log entries already present in its log, it ignores those entries in the new request.

5.6 Timing and availability

One of our requirements for Raft is that safety must not depend on timing: the system must not produce incorrect results just because some event happens more quickly or slowly than expected. However, availability (the ability of the system to respond to clients in a timely manner) must inevitably depend on timing. For example, if message exchanges take longer than the typical time between server crashes, candidates will not stay up long enough to win an election; without a steady leader, Raft cannot make progress.

Leader election is the aspect of Raft where timing is most critical. Raft will be able to elect and maintain a steady leader as long as the system satisfies the following *timing requirement*:

$$\mathit{broadcastTime} \ll \mathit{electionTimeout} \ll \mathit{MTBF}$$

In this inequality *broadcastTime* is the average time it takes a server to send RPCs in parallel to every server in the cluster and receive their responses; *electionTimeout* is the election timeout described in Section 5.2; and *MTBF* is the average time between failures for a single server. The broadcast time should be an order of magnitude less than the election timeout so that leaders can reliably send the heartbeat messages required to keep followers from starting elections; given the randomized approach used for election timeouts, this inequality also makes split votes unlikely. The election timeout should be a few orders of magnitude less than MTBF so that the system makes steady progress. When the leader crashes, the system will be unavailable for roughly the election timeout; we would like this to represent only a small fraction of overall time.

The broadcast time and MTBF are properties of the underlying system, while the election timeout is something we must choose. Raft's RPCs typically require the recipient to persist information to stable storage, so the broadcast time may range from 0.5ms to 20ms, depending on storage technology. As a result, the election timeout is likely to be somewhere between 10ms and 500ms. Typical server MTBFs are several months or more, which easily satisfies the timing requirement.

6 Cluster membership changes

Up until now we have assumed that the cluster *configuration* (the set of servers participating in the consensus algorithm) is fixed. In practice, it will occasionally be necessary to change the configuration, for example to replace servers when they fail or to change the degree of replication. Although this can be done by taking the entire cluster off-line, updating configuration files, and then restarting the cluster, this would leave the cluster unavailable during the changeover. In addition, if there are any manual steps, they risk operator error. In order to avoid these issues, we decided to automate configuration changes and incorporate them into the Raft consensus algorithm.

For the configuration change mechanism to be safe, there must be no point during the transition where it is possible for two leaders to be elected for the same term. Unfortunately, any approach where servers switch directly from the old configuration to the new configuration is unsafe. It isn't possible to atomically switch all of the servers at once, so the cluster can potentially split into two independent majorities during the transition (see Figure 10).

In order to ensure safety, configuration changes must use a two-phase approach. There are a variety of ways to implement the two phases. For example, some systems (e.g., [20]) use the first phase to disable the old configuration so it cannot process client requests; then the second phase enables the new configuration. In Raft the cluster first switches to a transitional configuration we call *joint consensus*; once the joint consensus has been committed, the system then transitions to the new configuration. The joint consensus combines both the old and new configurations:

- Log entries are replicated to all servers in both configurations.
- Any server from either configuration may serve as leader.
- Agreement (for elections and entry commitment) requires separate majorities from *both* the old and new configurations.

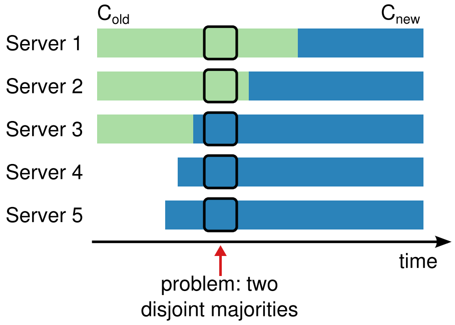


Figure 10: Switching directly from one configuration to another is unsafe because different servers will switch at different times. In this example, the cluster grows from three servers to five. Unfortunately, there is a point in time where two different leaders can be elected for the same term, one with a majority of the old configuration (C_{old}) and another with a majority of the new configuration (C_{new}).

The joint consensus allows individual servers to transition between configurations at different times without compromising safety. Furthermore, joint consensus allows the cluster to continue servicing client requests throughout the configuration change.

Cluster configurations are stored and communicated using special entries in the replicated log; Figure 11 illustrates the configuration change process. When the leader receives a request to change the configuration from C_{old} to C_{new}, it stores the configuration for joint consensus ($C_{old,new}$ in the figure) as a log entry and replicates that entry using the mechanisms described previously. Once a given server adds the new configuration entry to its log, it uses that configuration for all future decisions (a server always uses the latest configuration in its log, regardless of whether the entry is committed). This means that the leader will use the rules of $C_{old,new}$ to determine when the log entry for $C_{old,new}$ is committed. If the leader crashes, a new leader may be chosen under either C_{old} or $C_{old,new}$, depending on whether the winning candidate has received $C_{old,new}$. In any case, C_{new} cannot make unilateral decisions during this period.

Once $C_{old,new}$ has been committed, neither C_{old} nor C_{new} can make decisions without approval of the other, and the Leader Completeness Property ensures that only servers with the $C_{old,new}$ log entry can be elected as leader. It is now safe for the leader to create a log entry describing C_{new} and replicate it to the cluster. Again, this configuration will take effect on each server as soon as it is seen. When the new configuration has been committed under the rules of C_{new}, the old configuration is irrelevant and servers not in the new configuration can be shut down. As

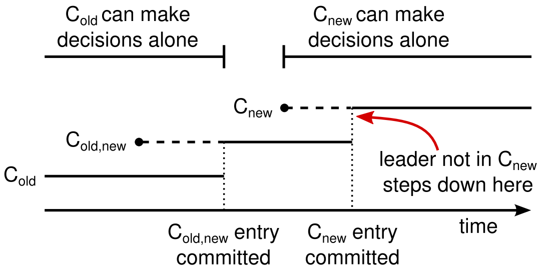

C_{old} can make decisions alone
C_{new} can make decisions alone
C_{new}
$C_{old,new}$
C_{old}
leader not in C_{new} steps down here
$C_{old,new}$ entry committed
C_{new} entry committed
time

Figure 11: Timeline for a configuration change. Dashed lines show configuration entries that have been created but not committed, and solid lines show the latest committed configuration entry. The leader first creates the $C_{\text{old,new}}$ configuration entry in its log and commits it to $C_{\text{old,new}}$ (a majority of C_{old} and a majority of C_{new}). Then it creates the C_{new} entry and commits it to a majority of C_{new}. There is no point in time in which C_{old} and C_{new} can both make decisions independently.

shown in Figure 11, there is no time when C_{old} and C_{new} can both make unilateral decisions; this guarantees safety.

There are three more issues to address for reconfiguration. The first issue is that new servers may not initially store any log entries. If they are added to the cluster in this state, it could take quite a while for them to catch up, during which time it might not be possible to commit new log entries. In order to avoid availability gaps, Raft introduces an additional phase before the configuration change, in which the new servers join the cluster as non-voting members (the leader replicates log entries to them, but they are not considered for majorities). Once the new servers have caught up with the rest of the cluster, the reconfiguration can proceed as described above.

The second issue is that the cluster leader may not be part of the new configuration. In this case, the leader steps down (returns to follower state) once it has committed the C_{new} log entry. This means that there will be a period of time (while it is committing C_{new}) when the leader is managing a cluster that does not include itself; it replicates log entries but does not count itself in majorities. The leader transition occurs when C_{new} is committed because this is the first point when the new configuration can operate independently (it will always be possible to choose a leader from C_{new}). Before this point, it may be the case that only a server from C_{old} can be elected leader.

The third issue is that removed servers (those not in C_{new}) can disrupt the cluster. These servers will not receive heartbeats, so they will time out and start new elections. They will then send RequestVote RPCs with new term numbers, and this will cause the current leader to revert to follower state. A new leader will eventually be elected, but the removed servers will time out again and the process will repeat, resulting in poor availability.

To prevent this problem, servers disregard RequestVote RPCs when they believe a current leader exists. Specifically, if a server receives a RequestVote RPC within the minimum election timeout of hearing from a current leader, it does not update its term or grant its vote. This does not affect normal elections, where each server waits at least a minimum election timeout before starting an election. However, it helps avoid disruptions from removed servers: if a leader is able to get heartbeats to its cluster, then it will not be deposed by larger term numbers.

7 Clients and log compaction

This section has been omitted due to space limitations, but the material is available in the extended version of this paper [29]. It describes how clients interact with Raft, including how clients find the cluster leader and how Raft supports linearizable semantics [8]. The extended version also describes how space in the replicated log can be reclaimed using a snapshotting approach. These issues apply to all consensus-based systems, and Raft's solutions are similar to other systems.

8 Implementation and evaluation

We have implemented Raft as part of a replicated state machine that stores configuration information for RAMCloud [30] and assists in failover of the RAMCloud coordinator. The Raft implementation contains roughly 2000 lines of C++ code, not including tests, comments, or blank lines. The source code is freely available [21]. There are also about 25 independent third-party open source implementations [31] of Raft in various stages of development, based on drafts of this paper. Also, various companies are deploying Raft-based systems [31].

The remainder of this section evaluates Raft using three criteria: understandability, correctness, and performance.

8.1 Understandability

To measure Raft's understandability relative to Paxos, we conducted an experimental study using upper-level undergraduate and graduate students in an Advanced Operating Systems course at Stanford University and a Distributed Computing course at U.C. Berkeley. We recorded a video lecture of Raft and another of Paxos, and created corresponding quizzes. The Raft lecture covered the content of this paper; the Paxos lecture covered enough material to create an equivalent replicated state machine, including single-decree Paxos, multi-decree Paxos, reconfiguration, and a few optimizations needed in practice (such as leader election). The quizzes tested basic understanding of the algorithms and also required students to reason about corner cases. Each student watched one video, took the corresponding quiz, watched the second video, and took the second quiz. About half of the participants did the Paxos portion first and the other half did the Raft portion first in order to account for both individual differences in performance and experience gained from the first portion of the study. We compared participants' scores on each quiz to determine whether participants showed a better understanding of Raft.

We tried to make the comparison between Paxos and

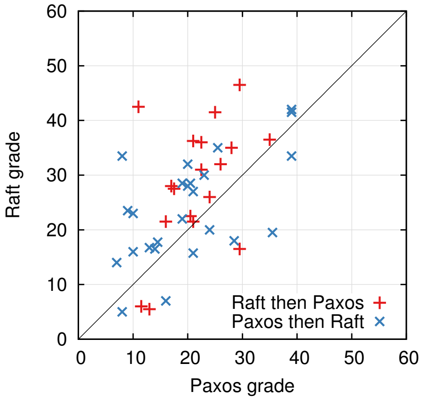


Figure 12: A scatter plot comparing 43 participants' performance on the Raft and Paxos quizzes. Points above the diagonal (33) represent participants who scored higher for Raft.

Raft as fair as possible. The experiment favored Paxos in two ways: 15 of the 43 participants reported having some prior experience with Paxos, and the Paxos video is 14% longer than the Raft video. As summarized in Table 1, we have taken steps to mitigate potential sources of bias. All of our materials are available for review [26, 28].

On average, participants scored 4.9 points higher on the Raft quiz than on the Paxos quiz (out of a possible 60 points, the mean Raft score was 25.7 and the mean Paxos score was 20.8); Figure 12 shows their individual scores. A paired *t*-test states that, with 95% confidence, the true distribution of Raft scores has a mean at least 2.5 points larger than the true distribution of Paxos scores.

We also created a linear regression model that predicts a new student's quiz scores based on three factors: which quiz they took, their degree of prior Paxos experience, and the order in which they learned the algorithms. The model predicts that the choice of quiz produces a 12.5-point difference in favor of Raft. This is significantly higher than the observed difference of 4.9 points, because many of the actual students had prior Paxos experience, which helped Paxos considerably, whereas it helped Raft slightly less. Curiously, the model also predicts scores 6.3 points lower on Raft for people that have already taken the Paxos quiz; although we don't know why, this does appear to be statistically significant.

We also surveyed participants after their quizzes to see which algorithm they felt would be easier to implement or explain; these results are shown in Figure 13. An overwhelming majority of participants reported Raft would be easier to implement and explain (33 of 41 for each question). However, these self-reported feelings may be less reliable than participants' quiz scores, and participants may have been biased by knowledge of our hypothesis that Raft is easier to understand.

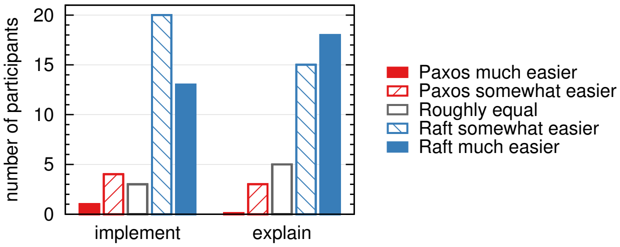


Figure 13: Using a 5-point scale, participants were asked (left) which algorithm they felt would be easier to implement in a functioning, correct, and efficient system, and (right) which would be easier to explain to a CS graduate student.

A detailed discussion of the Raft user study is available at [28].

8.2 Correctness

We have developed a formal specification and a proof of safety for the consensus mechanism described in Section 5. The formal specification [28] makes the information summarized in Figure 2 completely precise using the TLA+ specification language [15]. It is about 400 lines long and serves as the subject of the proof. It is also useful on its own for anyone implementing Raft. We have mechanically proven the Log Completeness Property using the TLA proof system [6]. However, this proof relies on invariants that have not been mechanically checked (for example, we have not proven the type safety of the specification). Furthermore, we have written an informal proof [28] of the State Machine Safety property which is complete (it relies on the specification alone) and relatively precise (it is about 3500 words long).

8.3 Performance

Raft's performance is similar to other consensus algorithms such as Paxos. The most important case for performance is when an established leader is replicating new log entries. Raft achieves this using the minimal number of messages (a single round-trip from the leader to half the cluster). It is also possible to further improve Raft's performance. For example, it easily supports batching and pipelining requests for higher throughput and lower latency. Various optimizations have been proposed in the literature for other algorithms; many of these could be applied to Raft, but we leave this to future work.

We used our Raft implementation to measure the performance of Raft's leader election algorithm and answer two questions. First, does the election process converge

Concern	Steps taken to mitigate bias	Materials for review [26, 28]
Equal lecture quality	Same lecturer for both. Paxos lecture based on and improved from existing materials used in several universities. Paxos lecture is 14% longer.	videos
Equal quiz difficulty	Questions grouped in difficulty and paired across exams.	quizzes
Fair grading	Used rubric. Graded in random order, alternating between quizzes.	rubric

Table 1: Concerns of possible bias against Paxos in the study, steps taken to counter each, and additional materials available.

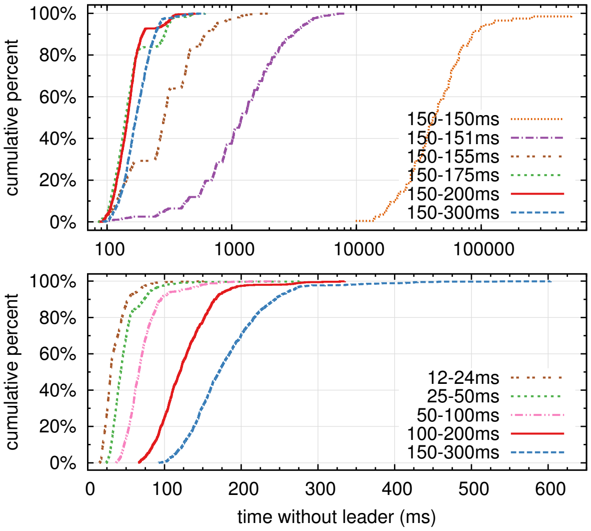


Figure 14: The time to detect and replace a crashed leader. The top graph varies the amount of randomness in election timeouts, and the bottom graph scales the minimum election timeout. Each line represents 1000 trials (except for 100 trials for "150–150ms") and corresponds to a particular choice of election timeouts; for example, "150–155ms" means that election timeouts were chosen randomly and uniformly between 150ms and 155ms. The measurements were taken on a cluster of five servers with a broadcast time of roughly 15ms. Results for a cluster of nine servers are similar.

quickly? Second, what is the minimum downtime that can be achieved after leader crashes?

To measure leader election, we repeatedly crashed the leader of a cluster of five servers and timed how long it took to detect the crash and elect a new leader (see Figure 14). To generate a worst-case scenario, the servers in each trial had different log lengths, so some candidates were not eligible to become leader. Furthermore, to encourage split votes, our test script triggered a synchronized broadcast of heartbeat RPCs from the leader before terminating its process (this approximates the behavior of the leader replicating a new log entry prior to crashing). The leader was crashed uniformly randomly within its heartbeat interval, which was half of the minimum election timeout for all tests. Thus, the smallest possible downtime was about half of the minimum election timeout.

The top graph in Figure 14 shows that a small amount of randomization in the election timeout is enough to avoid split votes in elections. In the absence of randomness, leader election consistently took longer than 10 seconds in our tests due to many split votes. Adding just 5ms of randomness helps significantly, resulting in a median downtime of 287ms. Using more randomness improves worst-case behavior: with 50ms of randomness the worst-case completion time (over 1000 trials) was 513ms.

The bottom graph in Figure 14 shows that downtime can be reduced by reducing the election timeout. With an election timeout of 12–24ms, it takes only 35ms on average to elect a leader (the longest trial took 152ms). However, lowering the timeouts beyond this point violates Raft's timing requirement: leaders have difficulty broadcasting heartbeats before other servers start new elections. This can cause unnecessary leader changes and lower overall system availability. We recommend using a conservative election timeout such as 150–300ms; such timeouts are unlikely to cause unnecessary leader changes and will still provide good availability.

9 Related work

There have been numerous publications related to consensus algorithms, many of which fall into one of the following categories:

- Lamport's original description of Paxos [13], and attempts to explain it more clearly [14, 18, 19].
- Elaborations of Paxos, which fill in missing details and modify the algorithm to provide a better foundation for implementation [24, 35, 11].
- Systems that implement consensus algorithms, such as Chubby [2, 4], ZooKeeper [9, 10], and Spanner [5]. The algorithms for Chubby and Spanner have not been published in detail, though both claim to be based on Paxos. ZooKeeper's algorithm has been published in more detail, but it is quite different from Paxos.
- Performance optimizations that can be applied to Paxos [16, 17, 3, 23, 1, 25].
- Oki and Liskov's Viewstamped Replication (VR), an alternative approach to consensus developed around the same time as Paxos. The original description [27] was intertwined with a protocol for distributed transactions, but the core consensus protocol has been separated in a recent update [20]. VR uses a leader-based approach with many similarities to Raft.

The greatest difference between Raft and Paxos is Raft's strong leadership: Raft uses leader election as an essential part of the consensus protocol, and it concentrates as much functionality as possible in the leader. This approach results in a simpler algorithm that is easier to understand. For example, in Paxos, leader election is orthogonal to the basic consensus protocol: it serves only as a performance optimization and is not required for achieving consensus. However, this results in additional mechanism: Paxos includes both a two-phase protocol for basic consensus and a separate mechanism for leader election. In contrast, Raft incorporates leader election directly into the consensus algorithm and uses it as the first of the two phases of consensus. This results in less mechanism than in Paxos.

Like Raft, VR and ZooKeeper are leader-based and therefore share many of Raft's advantages over Paxos. However, Raft has less mechanism that VR or ZooKeeper because it minimizes the functionality in non-leaders. For example, log entries in Raft flow in only one direction: outward from the leader in AppendEntries RPCs. In VR log entries flow in both directions (leaders can receive log entries during the election process); this results in additional mechanism and complexity. The published description of ZooKeeper also transfers log entries both to and from the leader, but the implementation is apparently more like Raft [32].

Raft has fewer message types than any other algorithm for consensus-based log replication that we are aware of. For example, VR and ZooKeeper each define 10 different message types, while Raft has only 4 message types (two RPC requests and their responses). Raft's messages are a bit more dense than the other algorithms', but they are simpler collectively. In addition, VR and ZooKeeper are described in terms of transmitting entire logs during leader changes; additional message types will be required to optimize these mechanisms so that they are practical.

Several different approaches for cluster membership changes have been proposed or implemented in other work, including Lamport's original proposal [13], VR [20], and SMART [22]. We chose the joint consensus approach for Raft because it leverages the rest of the consensus protocol, so that very little additional mechanism is required for membership changes. Lamport's α-based approach was not an option for Raft because it assumes consensus can be reached without a leader. In comparison to VR and SMART, Raft's reconfiguration algorithm has the advantage that membership changes can occur without limiting the processing of normal requests; in contrast, VR stops all normal processing during configuration changes, and SMART imposes an α-like limit on the number of outstanding requests. Raft's approach also adds less mechanism than either VR or SMART.

10 Conclusion

Algorithms are often designed with correctness, efficiency, and/or conciseness as the primary goals. Although these are all worthy goals, we believe that understandability is just as important. None of the other goals can be achieved until developers render the algorithm into a practical implementation, which will inevitably deviate from and expand upon the published form. Unless developers have a deep understanding of the algorithm and can create intuitions about it, it will be difficult for them to retain its desirable properties in their implementation.

In this paper we addressed the issue of distributed consensus, where a widely accepted but impenetrable algorithm, Paxos, has challenged students and developers for many years. We developed a new algorithm, Raft, which we have shown to be more understandable than Paxos. We also believe that Raft provides a better foundation for system building. Using understandability as the primary design goal changed the way we approached the design of Raft; as the design progressed we found ourselves reusing a few techniques repeatedly, such as decomposing the problem and simplifying the state space. These techniques not only improved the understandability of Raft but also made it easier to convince ourselves of its correctness.

11 Acknowledgments

The user study would not have been possible without the support of Ali Ghodsi, David Mazières, and the students of CS 294-91 at Berkeley and CS 240 at Stanford. Scott Klemmer helped us design the user study, and Nelson Ray advised us on statistical analysis. The Paxos slides for the user study borrowed heavily from a slide deck originally created by Lorenzo Alvisi. Special thanks go to David Mazières and Ezra Hoch for finding subtle bugs in Raft. Many people provided helpful feedback on the paper and user study materials, including Ed Bugnion, Michael Chan, Hugues Evrard, Daniel Giffin, Arjun Gopalan, Jon Howell, Vimalkumar Jeyakumar, Ankita Kejriwal, Aleksandar Kracun, Amit Levy, Joel Martin, Satoshi Matsushita, Oleg Pesok, David Ramos, Robbert van Renesse, Mendel Rosenblum, Nicolas Schiper, Deian Stefan, Andrew Stone, Ryan Stutsman, David Terei, Stephen Yang, Matei Zaharia, 24 anonymous conference reviewers (with duplicates), and especially our shepherd Eddie Kohler. Werner Vogels tweeted a link to an earlier draft, which gave Raft significant exposure. This work was supported by the Gigascale Systems Research Center and the Multiscale Systems Center, two of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the National Science Foundation under Grant No. 0963859, and by grants from Facebook, Google, Mellanox, NEC, NetApp, SAP, and Samsung. Diego Ongaro is supported by The Junglee Corporation Stanford Graduate Fellowship.

References

[1] Bolosky, W. J., Bradshaw, D., Haagens, R. B., Kusters, N. P., and Li, P. Paxos replicated state machines as the basis of a high-performance data store. In *Proc. NSDI'11, USENIX Conference on Networked Systems Design and Implementation* (2011), USENIX, pp. 141–154.

[2] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI'06, Symposium on Operating Systems Design and Implementation* (2006), USENIX, pp. 335–350.

[3] Camargos, L. J., Schmidt, R. M., and Pedone, F. Multicoordinated Paxos. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 316–317.

[4] Chandra, T. D., Griesemer, R., and Redstone, J. Paxos made live: an engineering perspective. In *Proc. PODC'07, ACM Symposium on Principles of Distributed Computing* (2007), ACM, pp. 398–407.

[5] 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 *Proc. OSDI'12, USENIX Conference on Operating Systems Design and Implementation* (2012), USENIX, pp. 251–264.

[6] Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., and Vanzetto, H. TLA$^+$ proofs. In *Proc. FM'12, Symposium on Formal Methods* (2012), D. Giannakopoulou and D. Méry, Eds., vol. 7436 of *Lecture Notes in Computer Science*, Springer, pp. 147–154.

[7] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proc. SOSP'03, ACM Symposium on Operating Systems Principles* (2003), ACM, pp. 29–43.

[8] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems 12* (July 1990), 463–492.

[9] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc ATC'10, USENIX Annual Technical Conference* (2010), USENIX, pp. 145–158.

[10] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *Proc. DSN'11, IEEE/IFIP Int'l Conf. on Dependable Systems & Networks* (2011), IEEE Computer Society, pp. 245–256.

[11] Kirsch, J., and Amir, Y. Paxos for system builders. Tech. Rep. CNDS-2008-2, Johns Hopkins University, 2008.

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commununications of the ACM 21*, 7 (July 1978), 558–565.

[13] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[14] Lamport, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[15] Lamport, L. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[16] Lamport, L. Generalized consensus and Paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, 2005.

[17] Lamport, L. Fast paxos. *Distributed Computing 19*, 2 (2006), 79–103.

[18] Lampson, B. W. How to build a highly available system using consensus. In *Distributed Algorithms*, O. Baboaglu and K. Marzullo, Eds. Springer-Verlag, 1996, pp. 1–17.

[19] Lampson, B. W. The ABCD's of Paxos. In *Proc. PODC'01, ACM Symposium on Principles of Distributed Computing* (2001), ACM, pp. 13–13.

[20] Liskov, B., and Cowling, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[21] LogCabin source code. `http://github.com/logcabin/logcabin`.

[22] Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. The SMART way to migrate replicated stateful services. In *Proc. EuroSys'06, ACM SIGOPS/EuroSys European Conference on Computer Systems* (2006), ACM, pp. 103–115.

[23] Mao, Y., Junqueira, F. P., and Marzullo, K. Mencius: building efficient replicated state machines for WANs. In *Proc. OSDI'08, USENIX Conference on Operating Systems Design and Implementation* (2008), USENIX, pp. 369–384.

[24] Mazières, D. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, Jan. 2007.

[25] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proc. SOSP'13, ACM Symposium on Operating System Principles* (2013), ACM.

[26] Raft user study. `http://ramcloud.stanford.edu/~ongaro/userstudy/`.

[27] Oki, B. M., and Liskov, B. H. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proc. PODC'88, ACM Symposium on Principles of Distributed Computing* (1988), ACM, pp. 8–17.

[28] Ongaro, D. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford University, 2014 (work in progress). `http://ramcloud.stanford.edu/~ongaro/thesis.pdf`.

[29] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm (extended version). `http://ramcloud.stanford.edu/raft.pdf`.

[30] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R. The case for RAMCloud. *Communications of the ACM 54* (July 2011), 121–130.

[31] Raft consensus algorithm website. `http://raftconsensus.github.io`.

[32] Reed, B. Personal communications, May 17, 2013.

[33] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[34] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. The Hadoop distributed file system. In *Proc. MSST'10, Symposium on Mass Storage Systems and Technologies* (2010), IEEE Computer Society, pp. 1–10.

[35] van Renesse, R. Paxos made moderately complex. Tech. rep., Cornell University, 2012.

GASPP: A GPU-Accelerated Stateful Packet Processing Framework

Giorgos Vasiliadis,[*] Lazaros Koromilas,[*] Michalis Polychronakis,[†] Sotiris Ioannidis[*]
[*]*FORTH-ICS,* [†]*Columbia University*
{gvasil,koromil,sotiris}@ics.forth.gr, mikepo@cs.columbia.edu

Abstract

Graphics processing units (GPUs) are a powerful platform for building high-speed network traffic processing applications using low-cost hardware. Existing systems tap the massively parallel architecture of GPUs to speed up certain computationally intensive tasks, such as cryptographic operations and pattern matching. However, they still suffer from significant overheads due to critical-path operations that are still being carried out on the CPU, and redundant inter-device data transfers.

In this paper we present GASPP, a programmable network traffic processing framework tailored to modern graphics processors. GASPP integrates optimized GPU-based implementations of a broad range of operations commonly used in network traffic processing applications, including the first purely GPU-based implementation of network flow tracking and TCP stream reassembly. GASPP also employs novel mechanisms for tackling control flow irregularities across SIMT threads, and sharing memory context between the network interface and the GPU. Our evaluation shows that GASPP can achieve multi-gigabit traffic forwarding rates even for computationally intensive and complex network operations such as stateful traffic classification, intrusion detection, and packet encryption. Especially when consolidating multiple network applications on the same device, GASPP achieves up to 16.2× speedup compared to standalone GPU-based implementations of the same applications.

1 Introduction

The emergence of commodity *many*-core architectures, such as multicore CPUs and modern graphics processors (GPUs) has proven to be a good solution for accelerating many network applications, and has led to their successful deployment in high-speed environments [10, 12–14, 26]. Recent trends have shown that certain network packet processing operations can be implemented efficiently on GPU architectures. Typically, such operations are either computationally intensive (e.g., encryption [14]), memory-intensive (e.g., IP routing [12]), or both (e.g., intrusion detection and prevention [13, 24, 26]). Modern GPU architectures offer high computational throughput and hide excessive memory latencies.

Unfortunately, the lack of programming abstractions and GPU-based libraries for network traffic processing—even for simple tasks such as packet decoding and filtering—increases significantly the programming effort needed to build, extend, and maintain high-performance GPU-based network applications. More complex critical-path operations, such as flow tracking and TCP stream reassembly, currently still run on the CPU, negatively offsetting any performance gains by the offloaded GPU operations. The absence of adequate OS support also increases the cost of data transfers between the host and I/O devices. For example, packets have to be transferred from the network interface to the user-space context of the application, and from there to kernel space in order to be transferred to the GPU. While programmers can explicitly optimize data movements, this increases the design complexity and code size of even simple GPU-based packet processing programs.

As a step towards tackling the above inefficiencies, we present *GASPP*, a network traffic processing framework tailored to modern graphics processors. GASPP integrates into a purely GPU-powered implementation many of the most common operations used by different types of network traffic processing applications, including the first GPU-based implementation of network flow tracking and TCP stream reassembly. By hiding complicated network processing issues while providing a rich and expressive interface that exposes only the data that matters to applications, GASPP allows developers to build complex GPU-based network traffic processing applications in a flexible and efficient way.

We have developed and integrated into GASPP novel mechanisms for sharing memory context between network interfaces and the GPU to avoid redundant data movement, and for scheduling packets in an efficient way that increases the utilization of the GPU and the shared PCIe bus. Overall, GASPP allows applications to scale in terms of performance, and carry out on the CPU only infrequently occurring operations.

The main contributions of our work are:

- We have designed, implemented, and evaluated GASPP, a novel GPU-based framework for high-performance network traffic processing, which eases the development of applications that process data at multiple layers of the protocol stack.
- We present the first (to the best of our knowledge) purely GPU-based implementation of flow state management and TCP stream reconstruction.
- We present a novel packet scheduling technique that tackles control flow irregularities and load imbalance across GPU threads.
- We present a zero-copy mechanism that avoids redundant memory copies between the network interface and the GPU, increasing significantly the throughput of cross-device data transfers.

2 Motivation

The Need for Modularity. The rise of general-purpose computing on GPUs (GPGPU) and related frameworks, such as CUDA and OpenCL, has made the implementation of GPU-accelerated applications easier than ever. Unfortunately, the majority of GPU-assisted network applications follow a monolithic design, lacking both modularity and flexibility. As a result, building, maintaining, and extending such systems eventually becomes a real burden. In addition, the absence of libraries for network processing operations—even for simple tasks like packet decoding or filtering—increases development costs even further. GASPP integrates a broad range of operations that different types of network applications rely on, with all the advantages of a GPU-powered implementation, into a single application development platform. This allows developers to focus on core application logic, alleviating the low-level technical challenges of data transfer to and from the GPU, packet batching, asynchronous execution, synchronization issues, connection state management, and so on.

The Need for Stateful Processing. Flow tracking and TCP stream reconstruction are mandatory features of a broad range of network applications. Intrusion detection and traffic classification systems typically inspect the application-layer stream to identify patterns that span multiple packets and thwart evasion attacks [9, 28]. Existing GPU-assisted network processing applications, however, just offload to the GPU certain data-parallel tasks, and are saturated by the many computationally heavy operations that are still being carried out on the CPU, such as network flow tracking, TCP stream reassembly, and protocol parsing [13, 26].

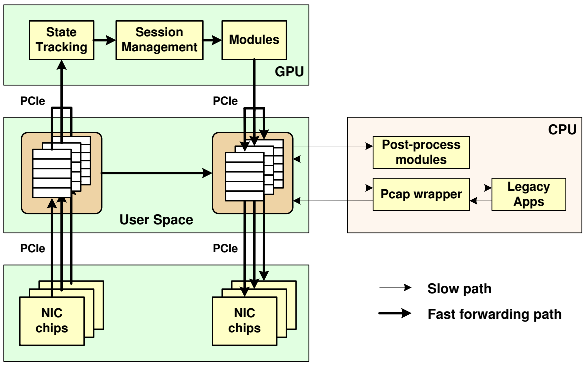


Figure 1: GASPP architecture.

The most common approach for stateful processing is to buffer incoming packets, reassemble them, and deliver "chunks" of the reassembled stream to higher-level processing elements [6, 7]. A major drawback of this approach is that it requires several data copies and significant extra memory space. In Gigabit networks, where packet intervals can be as short as 1.25 μsec (in a 10GbE network, for a MTU of 1.5KB), packet buffering requires large amounts of memory even for very short time windows. To address these challenges, the primary objectives of our GPU-based stateful processing implementation are: (i) process as many packets as possible *on-the-fly* (instead of buffering them), and (ii) ensure that packets of the same connection are processed *in-order*.

3 Design

The high-level design of GASPP is shown in Figure 1. Packets are transferred from the network interfaces to the memory space of the GPU in batches. The captured packets are then classified according to their protocol and are processed in parallel by the GPU. For stateful protocols, connection state management and TCP stream reconstruction are supported for delivering a consistent application-layer byte stream.

GASPP applications consist of *modules* that control all aspects of the traffic processing flow. Modules are represented as GPU device functions, and take as input a network packet or stream chunk. Internally, each module is executed in parallel on a batch of packets. After processing is completed, the packets are transferred back to the memory space of the host, and depending on the application, to the appropriate output network interface.

3.1 Processing Modules

A central concept of NVIDIA's CUDA [5] that has influenced the design of GASPP is the organization of GPU programs into *kernels*, which in essence are functions that are executed by groups of threads. GASPP allows

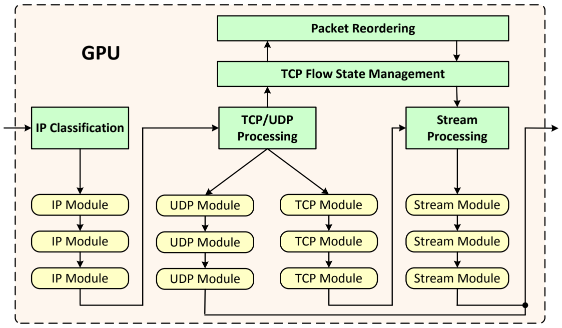


Figure 2: GPU packet processing pipeline. The pipeline is executed by a different thread for every incoming packet.

users to specify processing tasks on the incoming traffic by writing GASPP *modules*, applicable on different protocol layers, which are then mapped into GPU kernel functions. Modules can be implemented according to the following prototypes:

```
__device__ uint processEth(unsigned pktid,
    ethhdr *eth, uint cxtkey);
__device__ uint processIP(unsigned pktid,
    ethhdr *eth, iphdr *ip, uint cxtkey);
__device__ uint processUDP(unsigned pktid,
    ethhdr *eth, iphdr *ip, udphdr *udp, uint cxtkey);
__device__ uint processTCP(unsigned pktid,
    ethhdr *eth, iphdr *ip, tcphdr *tcp, uint cxtkey);
__device__ uint processStream(unsigned pktid,
    ethhdr *eth, iphdr *ip, tcphdr *tcp, uchar *chunk,
    unsigned chunklen, uint cxtkey);
```

The framework is responsible for decoding incoming packets and executing all registered `process*()` modules by passing the appropriate parameters. Packet decoding and stream reassembly is performed by the underlying system, eliminating any extra effort from the side of the developer. Each module is executed at the corresponding layer, with pointer arguments to the encapsulated protocol headers. Arguments also include a unique identifier for each packet and a user-defined key that denotes the packet's class (described in more detail in §5.3). Currently, GASPP supports the most common network protocols, such as Ethernet, IP, TCP and UDP. Other protocols can easily be handled by explicitly parsing raw packets. Modules are executed per-packet in a data-parallel fashion. If more than one modules have been registered, they are executed back-to-back in a packet processing pipeline, resulting in GPU module chains, as shown in Figure 2.

The `processStream()` modules are executed whenever a new normalized TCP chunk of data is available. These modules are responsible for keeping internally the state between consecutive chunks—or, alternatively, for storing chunks in global memory for future use—and continuing the processing from the last state of the previous chunk. For example, a pattern matching application can match the contents of the current chunk and keep the state of its matching algorithm to a global variable; on the arrival of the next chunk, the matching process will continue from the previously stored state.

As modules are simple to write, we expect that users will easily write new ones as needed using the function prototypes described above. In fact, the complete implementation of a module that simply passes packets from an input to an output interface takes only a few lines of code. More complex network applications, such as NIDS, L7 traffic classification, and packet encryption, require a few dozen lines of code, as described in §6.

3.2 API

To cover the needs of a broad range of network traffic processing applications, GASPP offers a rich GPU API with data structures and algorithms for processing network packets.

Shared Hash Table. GASPP enables applications to access the processed data through a global hash table. Data stored in an instance of the hash table is persistent across GPU kernel invocations, and is shared between the host and the device. Internally, data objects are hashed and mapped to a given bucket. To enable GPU threads to add or remove nodes from the table in parallel, we associate an atomic lock with each bucket, so that only a single thread can make changes to a given bucket at a time.

Pattern Matching. Our framework provides a GPU-based API for matching fixed strings and regular expressions. We have ported a variant of the Aho-Corasick algorithm for string searching, and use a DFA-based implementation for regular expression matching. Both implementations have linear complexity over the input data, independent of the number of patterns to be searched. To utilize efficiently the GPU memory subsystem, packet payloads are accessed 16-bytes at a time, using an `int4` variable [27].

Cipher Operations. Currently, GASPP provides AES (128-bit to 512-bit key sizes) and RSA (1024-bit and 2048-bit key sizes) functions for encryption and decryption, and supports all modes of AES (ECB, CTR, CFB and OFB). Again, packet contents are read and written 16-bytes at a time, as this substantially improves GPU performance. The encryption and decryption process happens in-place and as packet lengths may be modified, the checksums for IP and TCP/UDP packets are recomputed to be consistent. In cases where the NIC controller supports checksum computation offloading, GASPP simply forwards the altered packets to the NIC.

Network Packet Manipulation Functions. GASPP provides special functions for dropping network packets (`Drop()`), ignoring any subsequent registered user-defined modules (`Ignore()`), passing packets to the host for further processing (`ToLinux()`), or writing their contents to a dump file (`ToDump()`). Each function updates accordingly the packet index array, which holds the offsets where each packet is stored in the packet buffer, and a separate "metadata" array.

4 Stateful Protocol Analysis

The stateful protocol analysis component of GASPP is designed with minimal complexity so as to maximize processing speed. This component is responsible for maintaining the state of TCP connections, and reconstructing the application-level byte stream by merging packet payloads and reordering out-of-order packets.

4.1 Flow Tracking

GASPP uses a connection table array stored in the global device memory of the GPU for keeping the state of TCP connections. Each record is 17-byte long. A 4-byte hash of the source and destination IP addresses and TCP ports is used to handle collisions in the flow classifier. Connection state is stored in a single-byte variable. The sequence numbers of the most recently received client and server segments are stored in two 4-byte fields, and are updated every time the next in-order segment arrives. Hash table collisions are handled using a locking chained hash table with linked lists (described in detail in §3.2). A 4-byte pointer points to the next record (if any).

The connection table can easily fill up with adversarial partially-established connections, benign connections that stay idle for a long time, or connections that failed to terminate properly. For this reason, connection records that have been idle for more than a certain timeout, set to 60 seconds by default, are periodically removed. As current GPU devices do not provide support for measuring real-world time, we resort to a separate GPU kernel that is initiated periodically according to the timeout value. Its task is to simply mark each connection record by setting the first bit of the state variable. If a connection record is already marked, it is removed from the table. A marked record is unmarked when a new packet for this connection is received before the timeout expires.

4.2 Parallelizing TCP Stream Reassembly

Maintaining the state of incoming connections is simple as long as the packets that are processed in parallel by the GPU belong to different connections. Typically, however, a batch of packets usually contains several packets of the same connection. It is thus important to ensure that the order of connection updates will be correct when processing packets of the same connection in parallel.

TCP reconstruction threads are synchronized through a separate array used for pairing threads that must process consecutive packets. When a new batch is received, each thread hashes its packet twice: once using $hash(addr\_s, addr\_d, port\_s, port\_d, seq)$, and a second time using $hash(addr\_s, addr\_d, port\_s, port\_d, seq+len)$, as shown in Figure 3. A memory barrier is used to guarantee that all threads have finished hashing their packets. Using this scheme, two packets x and y are consecutive if: $hash_x(4\text{-}tuple, seq + len) = hash_y(4\text{-}tuple, seq)$. The hash function is unidirectional to ensure that each stream direction is reconstructed separately. The SYN and SYN-ACK packets are paired by hashing the sequence and acknowledge numbers correspondingly. If both the SYN and SYN-ACK packets are present, the state of the connection is changed to `ESTABLISHED`, otherwise if only the SYN packet is present, the state is set to `SYN_RECEIVED`.

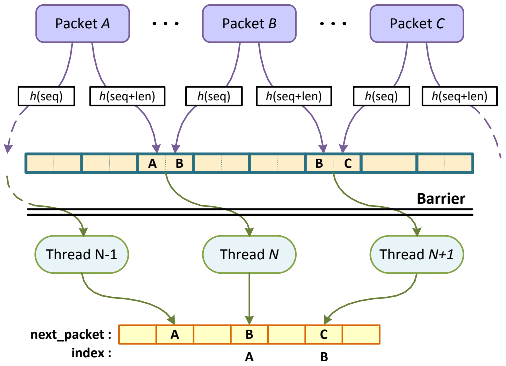


Figure 3: Ordering sequential TCP packets in parallel. The resulting *next_packet* array contains the next in-order packet, if any (i.e. *next_packet[A] = B*).

Having hashed all pairs of consecutive packets in the hash table, the next step is to create the proper packet ordering for each TCP stream using the `next_packet` array, as shown in Figure 3. Each packet is uniquely identified by an `id`, which corresponds to the index where the packet is stored in the packet index array. The `next_packet` array is set at the beginning of the current batch, and its cells contain the `id` of the next in-order packet (or `-1` if it does not exist in the current batch), e.g., if `x` is the `id` of the current packet, the `id` of the next in-order packet will be `y = next_packet[x]`. Finally, the connection table is updated with the sequence number of the last packet of each flow direction, i.e., the packet `x` that does not have a next packet in the current batch.

4.3 Packet Reordering

Although batch processing handles out-of-order packets that are included in the same batch, it does not solve the problem in the general case. A potential solution for in-line applications would be to just drop out-of-sequence packets, forcing the host to retransmit them. Whenever an expected packet would be missing, subsequent pack-

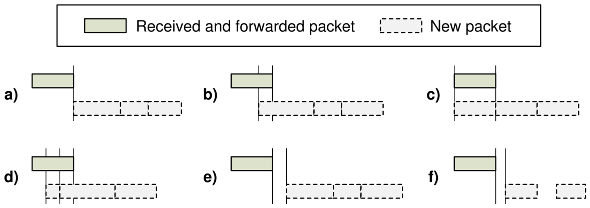


Figure 4: Subsequent packets (dashed line) may arrive in-sequence ((a)–(d)) or out of order, creating holes in the reconstructed TCP stream ((e)–(f)).

ets would be actively dropped until the missing packet arrives. Although this approach would ensure an in-order packet flow, it has several disadvantages. First, in situations where the percentage of out-of-order packets is high, performance will degrade. Second, if the endpoints are using selective retransmission and there is a high rate of data loss in the network, connections would be rendered unusable due to excessive packet drops.

To deal with TCP sequence hole scenarios, GASPP only processes packets with sequence numbers less than or equal to the connection's current sequence number (Figure 4(a)–(d)). Received packets with no preceding packets in the current batch and with sequence numbers larger than the ones stored in the connection table imply sequence holes (Figure 4(e)–(f)), and are copied in a separate buffer in global device memory. If a thread encounters an out-of-order packet (i.e., a packet with a sequence number larger than the sequence number stored in the connection table, with no preceding packet in the current batch after the hashing calculations of §4.2), it traverses the `next_packet` array and marks as out-of-order all subsequent packets of the same flow contained in the current batch (if any). This allows the system to identify sequences of out-of-order packets, as the ones shown in the examples of Figure 4(e)–(f). The buffer size is configurable and can be up to several hundred MBs, depending on the network needs. If the buffer contains any out-of-order packets, these are processed right after a new batch of incoming packets is processed.

Although packets are copied using the very fast device-to-device copy mechanism, with a memory bandwidth of about 145 GB/s, an increased number of out-of-order packets can have a major effect on overall performance. For this reason, by default we limit the number of out-of-order packets that can be buffered to be equal to the available slots in a batch of packets. This size is enough under normal conditions, where out-of-order packets are quite rare [9], and it can be configured as needed for other environments. If the percentage of out-of-order packets exceeds this limit, our system starts to drop out-of-order packets, causing the corresponding host to retransmit them.

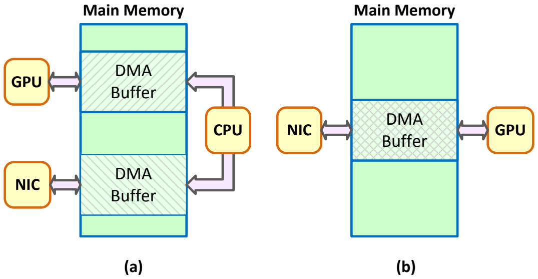


Figure 5: Normal (a) and zero-copy (b) data transfer between the NIC and the GPU.

5 Optimizing Performance

5.1 Inter-Device Data Transfer

The problem of data transfers between the CPU and the GPU is well-known in the GPGPU community, as it results in redundant cross-device communication. The traditional approach is to exchange data using DMA between the memory regions assigned by the OS to each device. As shown in Figure 5(a), network packets are transferred to the page-locked memory of the NIC, then copied to the page-locked memory of the GPU, and from there, they are finally transferred to the GPU.

To avoid costly packet copies and context switches, GASPP uses a single buffer for efficient data sharing between the NIC and the GPU, as shown in Figure 5(b), by adjusting the `netmap` module [20]. The shared buffer is added to the internal tracking mechanism of the CUDA driver to automatically accelerate calls to functions, as it can be accessed directly by the GPU. The buffer is managed by GASPP through the specification of a policy based on time and size constraints. This enables real-time applications to process incoming packets whenever a timeout is triggered, instead of waiting for buffers to fill up over a specified threshold. Per-packet buffer allocation overheads are reduced by transferring several packets at a time. Buffers consist of fixed-size slots, with each slot corresponding to one packet in the hardware queue. Slots are reused whenever the circular hardware queue wraps around. The size of each slot is 1,536 bytes, which is consistent with the NIC's alignment requirements, and enough for the typical 1,518-byte maximum Ethernet frame size.

Although making the NIC's packet queue directly accessible to the GPU eliminates redundant copies, this does not always lead to better performance. As previous studies have shown [12, 26] (we verify their results in §7.1), contrary to NICs, current GPU implementations suffer from poor performance for small data transfers. To improve PCIe throughput, we batch several packets and transfer them at once. However, the fixed-size partitioning of the NIC's queue leads to redundant data transfers for traffic with many small packets. For example, a 64-

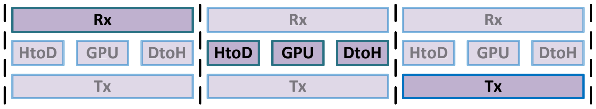


Figure 6: The I/O and processing pipeline.

byte packet consumes only 1/24th of the available space in its slot. This introduces an interesting trade-off, and as we show in §7.1, occasionally it is better to copy packets back-to-back into a second buffer and transferring it to the GPU. GASPP dynamically switches to the optimal approach by monitoring the actual utilization of the slots.

The forwarding path requires the transmission of network packets after processing is completed, and this is achieved using a triple-pipeline solution, as shown in Figure 6. Packet reception, GPU data transfers and execution, and packet transmission are executed asynchronously in a multiplexed manner.

5.2 Packet Decoding

Memory alignment is a major factor that affects the packet decoding process, as GPU execution constrains memory accesses to be aligned for all data types. For example, `int` variables should be stored to addresses that are a multiple of `sizeof(int)`. Due to the layered nature of network protocols, however, several fields of encapsulated protocols are not aligned when transferred to the memory space of the GPU. To overcome this issue, GASPP reads the packet headers from global memory, parses them using bitwise logic and shifting operations, and stores them in appropriately aligned structures. To optimize memory usage, input data is accessed in units of 16 bytes (using an `int4` variable).

5.3 Packet Scheduling

Registered modules are scheduled on the GPU, per protocol, in a serial fashion. Whenever a new batch of packets is available, it is processed in parallel using a number of threads equal to the number of packets in the batch (each thread processes a different packet). As shown in Figure 2, all registered modules for a certain protocol are executed serially on decoded packets in a lockstep way.

Network packets are processed by different threads, grouped together into logical units known as *warps* (in current NVIDIA GPU architectures, 32 threads form a warp) and mapped to SIMT units. As threads within the same warp have to execute the same instructions, load imbalance and code flow divergence within a warp can cause inefficiencies. This may occur under the following primary conditions: (i) when processing different transport-layer protocols (i.e., TCP and UDP) in the same warp, (ii) in full-packet processing applications when packet lengths within a warp differ significantly, and (iii) when different packets follow different processing paths, i.e., threads of the same warp execute different user-defined modules.

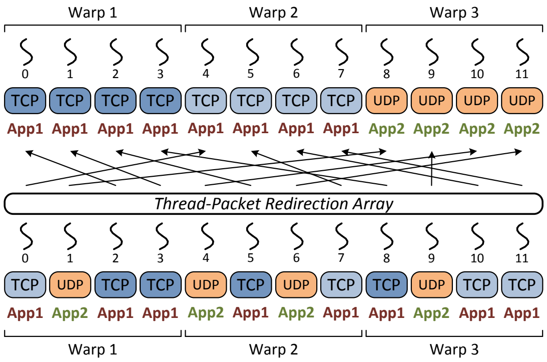

Warp 1
Warp 2
Warp 3
0
1
2
3
4
5
6
7
8
9
10
11
TCP TCP TCP TCP TCP TCP TCP TCP UDP UDP UDP UDP
App1 App1 App1 App1 App1 App1 App1 App1 App2 App2 App2 App2
Thread-Packet Redirection Array
0
1
2
3
4
5
6
7
8
9
10
11
TCP UDP TCP TCP UDP TCP UDP TCP TCP UDP TCP TCP
App1 App2 App1 App1 App2 App1 App2 App1 App1 App2 App1 App1
Warp 1
Warp 2
Warp 3

Figure 7: Packet scheduling for eliminating control flow divergences and load imbalances. Packet brightness represents packet size.

As the received traffic mix is typically very dynamic, it is essential to find an appropriate mapping between threads and network packets at runtime. It is also crucial that the overhead of the mapping process is low, so as to not jeopardize overall performance. To that end, our basic strategy is to group the packets of a batch according to their encapsulated transport-layer protocol and their length. In addition, module developers can specify *context keys* to describe packets that belong to the same class, which should follow the same module execution pipeline. A context key is a value returned by a user-defined module and is passed (as the final parameter) to the next registered module. GASPP uses these context keys to further pack packets of the same class together and map them to threads of the same warp after each module execution. This gives developers the flexibility to build complex packet processing pipelines that will be mapped efficiently to the underlying GPU architecture at runtime.

To group a batch of packets on the GPU, we have adapted a GPU-based radix sort implementation [1]. Specifically, we assign a separate weight for each packet consisting of the byte concatenation of the `ip_proto` field of its IP header, the value of the context key returned by the previously executed module, and its length. Weights are calculated on the GPU after each module execution using a separate thread for each packet, and are used by the radix sort algorithm to group the packets. Moreover, instead of copying each packet to the appropriate (i.e., sorted) position, we simply change their order in the packet index array. We also attempted to relocate packets by transposing the packet array on the GPU device memory, in order to benefit from memory coalescing [5]. Unfortunately, the overall cost of the corresponding data movements was not amortized by the resulting memory coalescing gains.

Using the above procedure, GASPP assigns dynamically to the same warp any similar-sized packets meant to be processed by the same module, as shown in Figure 7. Packets that were discarded earlier or of which the processing pipeline has been completed are grouped and mapped to warps that contain only idle threads—otherwise warps would contain both idle and active threads, degrading the utilization of the SIMT processors. To prevent packet reordering from taking place during packet forwarding, we also preserve the initial (pre-sorted) packet index array. In §7.2 we analyze in detail how control flow divergence affects the performance of the GPU, and show how our packet scheduling mechanisms tackle the irregular code execution at a fixed cost.

6 Developing with GASPP

In this section we present simple examples of representative applications built using the GASPP framework.

L3/L4 Firewall. Firewalls operate at the network layer (port-based) or the application layer (content-based). For our purposes, we have built a GASPP module that can drop traffic based on Layer-3 and Layer-4 rules. An incoming packet is filtered if the corresponding IP addresses and port numbers are found in the hash table; otherwise the packet is forwarded.

L7 Traffic Classification. We have implemented a L7 traffic classification tool (similar to the L7-filter tool [2]) on top of GASPP. The tool dynamically loads the pattern set of the L7-filter tool, in which each application-level protocol (HTTP, SMTP, etc.) is represented by a different regular expression. At runtime, each incoming flow is matched against each regular expression independently. In order to match patterns that cross TCP segment boundaries that lie on the same batch, each thread continues the processing to the next TCP segment (obtained from the `next_packet` array). The processing of the next TCP segment continues until a final or a fail DFA-state is reached, as suggested in [25]. In addition, the DFA-state of the last TCP segment of the current batch is stored in a global variable, so that on the arrival of the next stream chunk, the matching process continues from the previously stored state. This allows the detection of regular expressions that span (potentially deliberately) not only multiple packets, but also two or more stream chunks.

Signature-based Intrusion Detection. Modern NIDS, such as Snort [7], use a large number of regular expressions to determine whether a packet stream contains an attack vector or not. To reduce the number of packets that need to be matched against a regular expression, typical NIDS take advantage of the string matching engine and use it as a first-level filtering mechanism before proceeding to regular expression matching. We have implemented the same functionality on top of GASPP, using a different module for scanning each incoming traffic stream against all the fixed strings in a signature set. Patterns that cross TCP segments are handled similarly to the L7 Traffic Classification module. Only the matching streams are further processed against the corresponding regular expressions set.

Buffer	1KB	4KB	64KB	256KB	1MB	16MB
Host to GPU	2.04	7.12	34.4	42.1	45.7	47.8
GPU to Host	2.03	6.70	21.1	23.8	24.6	24.9

Table 1: Sustained PCIe throughput (Gbit/s) for transferring data to a single GPU, for different buffer sizes.

Packet size (bytes)	64	128	256	512	1024	1518
Copy back-to-back	13.76	18.21	20.53	19.21	19.24	20.04
Zero-copy	2.06	4.03	8.07	16.13	32.26	47.83

Table 2: Sustained throughput (Gbit/s) for various packet sizes, when bulk-transferring data to a single GPU.

AES. Encryption is used by protocols and services, such as SSL, VPN, and IPsec, for securing communications by authenticating and encrypting the IP packets of a communication session. While stock protocol suites that are used to secure communications, such as IPsec, actually use connectionless integrity and data origin authentication, for simplicity, we only encrypt all incoming packets using the AES-CBC algorithm and a different 128-bit key for each connection.

7 Performance Evaluation

Hardware Setup Our base system is equipped with two Intel Xeon E5520 Quad-core CPUs at 2.27GHz and 12 GB of RAM (6 GB per NUMA domain). Each CPU is connected to peripherals via a separate I/O hub, linked to several PCIe slots. Each I/O hub is connected to an NVIDIA GTX480 graphics card via a PCIe v2.0 x16 slot, and one Intel 82599EB with two 10 GbE ports, via a PCIe v2.0 8× slot. The system runs Linux 3.5 with CUDA v5.0 installed. After experimentation, we have found that the best placement is to have a GPU and a NIC on each NUMA node. We also place the GPU and NIC buffers in the same memory domain, as local memory accesses sustain lower latency and more bandwidth compared to remote accesses.

For traffic generation we use a custom packet generator built on top of `netmap` [20]. Test traffic consists of both synthetic traffic, as well as real traffic traces.

7.1 Data Transfer

We evaluate the zero-copy mechanism by taking into account the size of the transferred packets. The system can efficiently deliver all incoming packets to user space, regardless of the packet size, by mapping the NIC's DMA

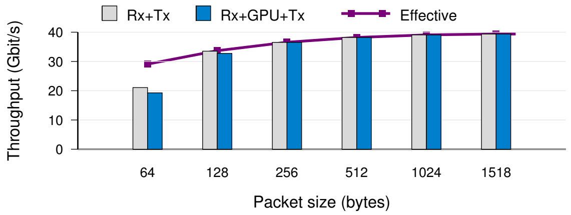


Figure 8: Data transfer throughput for different packet sizes when using two dual-port 10GbE NICs.

packet buffer. However, small data transfers to the GPU incur significant penalties. Table 1 shows that for transfers of less than 4KB, the PCIe throughput falls below 7 Gbit/s. With a large buffer though, the transfer rate to the GPU exceeds 45 Gbit/s, while the transfer rate from the GPU to the host decreases to about 25 Gbit/s.[1]

To overcome the low PCIe throughput, GASPP transfers batches of network packets to the GPU, instead of one at a time. However, as packets are placed in fixed-sized slots, transferring many slots at once results in redundant data transfers when the slots are not fully occupied. As we can see in Table 2, when traffic consists of small packets, the actual PCIe throughput drops drastically. Thus, it is better to copy small network packets sequentially into another buffer, rather than transfer the corresponding slots directly. Direct transfer pays off only for packet sizes over 512 bytes (when buffer occupancy is over $512/1536 = 33.3\%$), achieving 47.8 Gbit/s for 1518-byte packets (a 2.3× speedup).

Consequently, we adopted a simple *selective offloading* scheme, whereby packets in the shared buffer are copied to another buffer sequentially (in 16-byte aligned boundaries) if the overall occupancy of the shared buffer is sparse. Otherwise, the shared buffer is transferred directly to the GPU. Occupancy is computed—without any additional overhead—by simply counting the number of bytes of the newly arrived packets every time a new interrupt is generated by the NIC.

Figure 8 shows the throughput for forwarding packets with all data transfers included, but without any GPU computations. We observe that the forwarding performance for 64-byte packets reaches 21 Gbit/s, out of the maximum 29.09 Gbit/s, while for large packets it reaches the maximum full line rate. We also observe that the GPU transfers of large packets are completely hidden on the Rx+GPU+Tx path, as they are performed in parallel using the pipeline shown in Figure 6, and thus they do not affect overall performance. Unfortunately, this is not the case for small packets (less than 128-bytes), which suffer an additional 2–9% hit due to memory contention.

7.2 Raw GPU Processing Throughput

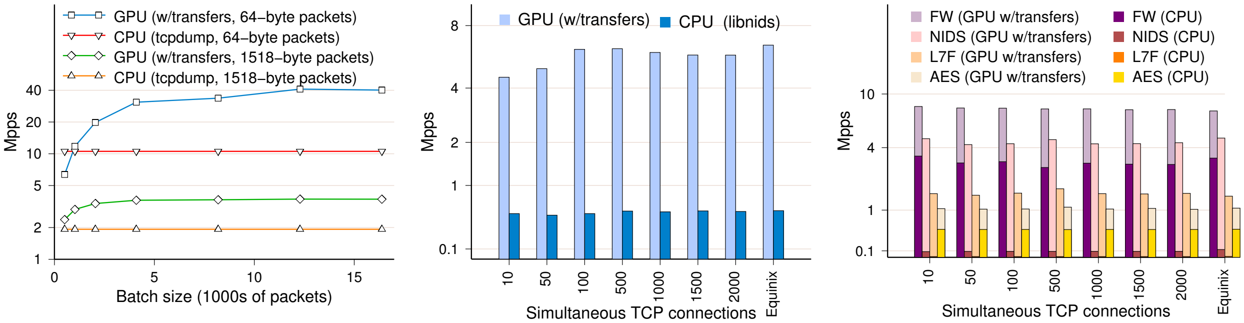
Having examined data transfer costs, we now evaluate the computational performance of a single GPU—exluding all network I/O transfers—for packet decoding, connection state management, TCP stream reassembly, and some representative traffic processing applications.

Packet Decoding. Decoding a packet according to its protocols is one of the most basic packet processing operations, and thus we use it as a base cost of our framework. Figure 9(a) shows the GPU performance for fully decoding incoming UDP packets into appropriately aligned structures, as described in §5.2 (throughput is very similar for TCP). As expected, the throughput increases as the number of packets processed in parallel increases. When decoding 64-byte packets, the GPU performance with PCIe transfers included, reaches 48 Mpps, which is about 4.5 times faster than the computational throughput of the `tcpdump` decoding process sustained by a single CPU core, when packets are read from memory. For 1518-byte packets, the GPU sustains about 3.8 Mpps and matches the performance of 1.92 CPU cores.

Connection State Management and TCP Stream Reassembly. In this experiment we measure the performance of maintaining connection state on the GPU, and the performance of reassembling the packets of TCP flows into application-level streams. Figure 9(b) shows the packets processed per second for both operations. Test traffic consists of real HTTP connections with random IP addresses and TCP ports. Each connection fetches about 800KB from a server, and comprises about 870 packets (320 minimum-size ACKs, and 550 full-size data packets). We also use a trace-driven workload (“Equinix”) based on a trace captured by CAIDA’s *equinix-sanjose* monitor [3], in which the average and median packet length is 606.2 and 81 bytes respectively.

Keeping state and reassembling streams requires several hashtable lookups and updates, which result to marginal overhead for a sufficient number of simultaneous TCP connections and the Equinix trace; about 20–25% on the raw GPU performance sustained for packet decoding, that increases to 45–50% when the number of concurrent connections is low. The reason is that smaller numbers of concurrent connections result to lower parallelism. To compare with a CPU implementation, we measure the equivalent functionality of the Libnids TCP reassembly library [6], when packets are read from memory. Although Libnids implements more specific cases of the TCP stack processing, compared to GASPP, the network traces that we used for the evaluation enforce exactly the same functionality to be exercised. We can see that the throughput of a single CPU core is 0.55 Mpps, about 10× lower than the GPU version with all PCIe data transfers included.

[1] The PCIe asymmetry in the data transfer throughput is related to the interconnection between the motherboard and the GPUs [12].



(a) Packet decode.

(b) TCP state management and reassembly

(c) Applications.

Figure 9: Average processing throughput sustained by the GPU to (a) decode network packets, (b) maintain flow state and reassemble TCP streams, and (c) perform various network processing operations.

Elements	1M buckets	8M buckets	16M buckets
0.1M	463	3,595	7,166
1M	463	3,588	7,173
2M	934	3,593	7,181
4M	1,924	3,593	7,177
8M	3,935	3,597	7,171
16M	7,991	7,430	7,173
32M	16,060	15,344	14,851

Table 3: Time spent (μsec) for traversing the connection table and removing expired connections.

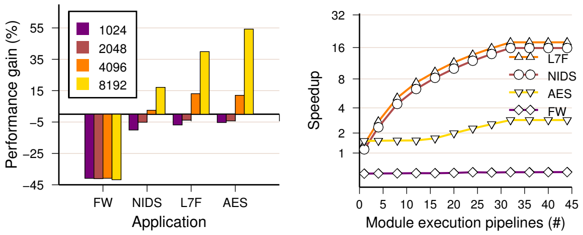
Removing Expired Connections. Removal of expired connections is very important for preventing the connection table from becoming full with stale adversarial connections, idle benign connections, or connections that failed to terminate cleanly [28]. Table 3 shows the GPU time spent for connection expiration. The time spent to traverse the table is constant when occupancy is lower than 100%, and analogous to the number of buckets; for larger values it increases due to the extra overhead of iterating the chain lists. Having a small hash table with a large load factor is better than a large but sparsely populated table. For example, the time to traverse a 1M-bucket table that contains up to 1M elements is about 20× lower than a 16M-bucket table with the same number of elements. If the occupancy is higher than 100% though, it is slightly better to use a 16M-bucket table.

Packet Processing Applications. In this experiment we measure the computational throughput of the GPU for the applications presented in §6. The NIDS is configured to use all the `content` patterns (about 10,000 strings) of the latest Snort distribution [7], combined into a single Aho-Corasick state machine, and their corresponding `pcre` regular expressions compiled into individual DFA state machines. The application-layer filter application (L7F) uses the "best-quality" patterns (12 regular expressions for identifying common services such as HTTP and SSH) of L7-filter [2], compiled into 12 different DFA state machines. The Firewall (FW) application uses 10,000 randomly generated rules for blocking incoming and outgoing traffic based on certain TCP/UDP port numbers and IP addresses. The test traffic consists of the HTTP-based traffic and the trace-driven Equinix workload described earlier. Note that the increased asymmetry in packet lengths and network protocols in the above traces is a stress-test workload for our data-parallel applications, given the SIMT architecture of GPUs [5].

Figure 9(c) shows the GPU throughput sustained by each application, including PCIe transfers, when packets are read from host memory. FW, as expected, has the highest throughput of about 8 Mpps—about 2.3 times higher than the equivalent single-core CPU execution. The throughput for NIDS is about 4.2–5.7 Mpps, and for L7F is about 1.45–1.73 Mpps. The large difference between the two applications is due to the fact that the NIDS shares the same Aho-Corasick state machine to initially search all packets (as we described in §6). In the common case, each packet will be matched only once against a single DFA. In contrast, the L7F requires each packet to be explicitly matched against each of the 12 different regular expression DFAs for both CPU and GPU implementations. The corresponding single-core CPU implementation of NIDS reaches about 0.1 Mpps, while L7F reaches 0.01 Mpps. We also note that both applications are explicitly forced to match all packets of all flows, even after they have been successfully classified (worst-case analysis). Finally, AES has a throughput of about 1.1 Mpps, as it is more computationally intensive. The corresponding CPU implementation using the AES-NI [4] instruction set on a single core reaches about 0.51 Mpps.[2]

Packet Scheduling In this experiment we measure how the packet scheduling technique, described in §5.3,

[2] The CPU performance of AES was measured on an Intel Xeon E5620 at 2.40GHz, because the Intel Xeon E5520 of our base system does not support AES-NI.



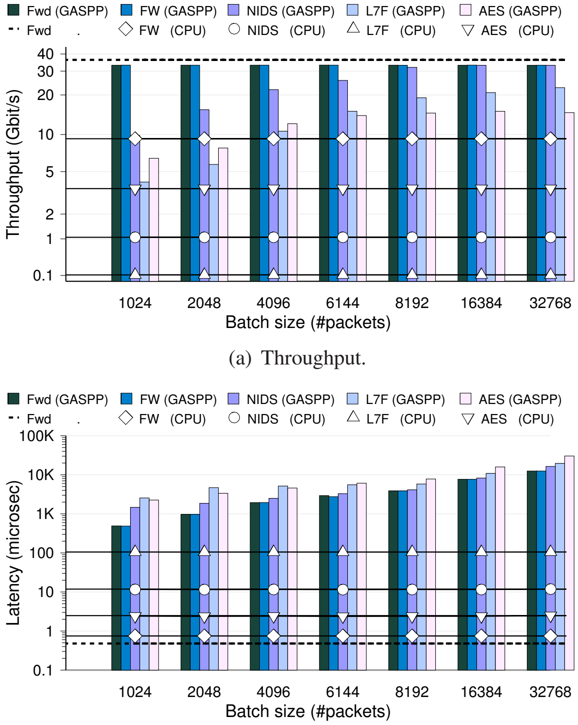
(a) Load imbalance (b) Divergence

Figure 10: Performance gains on raw GPU execution time when applying packet scheduling (the scheduling cost is included).

affects the performance of different network applications. For test traffic we used the trace-driven Equinix workload. Figure 10(a) shows the performance gain of each application for different packet batch sizes. We note that although the actual work of the modules is the same every time (i.e., the same processing will be applied on each packet), it is executed by different code blocks, thus execution is forced to diverge.

We observe that packet scheduling boosts the performance of full-packet processing applications, up to 55% for computationally intensive workloads like AES. Memory-intensive applications, such as NIDS, have a lower (about 15%) benefit. We also observe that gains increase as the batch size increases. With larger batch sizes, there is a greater range of packet sizes and protocols, hence more opportunities for better grouping. In contrast, packet scheduling has a negative effect on lightweight processing (as in FW, which only processes a few bytes of each packet), because the sorting overhead is not amortized by the resulting SIMT execution. As we cannot know at runtime if processing will be heavyweight or not, it is not feasible to predict if packet scheduling is worth applying. As a result, quite lightweight workloads (as in FW) will perform worse, although this lower performance will be hidden most of the time by data transfer overlap (Figure 6).

Another important aspect is how control flow divergence affects performance, e.g., when packets follow different module execution pipelines. To achieve this, we explicitly enforce different packets of the same batch to be processed by different modules. Figure 10(b) shows the achieved speedup when applying packet scheduling over the baseline case of mapping packets to thread warps without any reordering (network order). We see that as the number of different modules increases, our packet scheduling technique achieves a significant speedup. The speedup stabilizes after the number of modules exceeds 32, as only 32 threads (warp size) can run in a SIMT manner any given time. In general, code divergence within warps plays a significant role in GPU performance. The thread remapping achieved through our packet scheduling technique tolerates the irregular code execution at a fixed cost.



(a) Throughput.

(b) Latency.

Figure 11: Sustained traffic forwarding throughput (a) and latency (b) for GASPP-enabled applications.

7.3 End-to-End Performance

Individual Applications. Figure 11 shows the sustained end-to-end forwarding throughput and latency of individual GASPP-enabled applications for different batch sizes. We use four different traffic generators, equal to the number of available 10 GbE ports in our system. To prevent synchronization effects between the generators, the test workload consists of the HTTP-based traffic described earlier. For comparison, we also evaluate the corresponding CPU-based implementations running on a single core, on top of `netmap`.

The FW application can process all traffic delivered to the GPU, even for small batch sizes. NIDS, L7F, and AES, on the other hand, require larger batch sizes. The NIDS application requires batches of 8,192 packets to reach similar performance. Equivalent performance would be achieved (assuming ideal parallelization) by 28.4 CPU cores. More computationally intensive applications, however, such as L7F and AES, cannot process all traffic. L7F reaches 19 Gbit/s a batch size of 8,192 packets, and converges to 22.6 Gbit/s for larger sizes—about 205.1 times faster than a single CPU core. AES converges to about 15.8 Gbit/s, and matches the perfor-

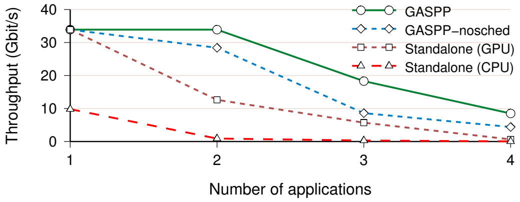


Figure 12: Sustained throughput for concurrently running applications.

mance of 4.4 CPU cores with AES-NI support. As expected, latency increases linearly with the batch size, and for certain applications and large batch sizes it can reach tens of milliseconds (Figure 11(b)). Fortunately, a batch size of 8,192 packets allows for a reasonable latency for all applications, while it sufficiently utilizes the PCIe bus and the parallel capabilities of the GTX480 card (Figure 11(a)). For instance, NIDS, L7F, and FW have a latency of 3–5 ms, while AES, which suffers from an extra GPU-to-host data transfer, has a latency of 7.8 ms.

Consolidated Applications. Consolidating multiple applications has the benefit of distributing the overhead of data transfer, packet decoding, state management, and stream reassembly across all applications, as all these operations are performed only once. Moreover, through the use of context keys, GASPP optimizes SIMT execution when packets of the same batch are processed by different applications. Figure 12 shows the sustained throughput when running multiple GASPP applications. Applications are added in the following order: FW, NIDS, L7F, AES (increasing overhead). We also enforce packets of different connections to follow different application processing paths. Specifically, we use the hash of the each packet's 5-tuple for deciding the order of execution. For example, a class of packets will be processed by application 1 and then application 2, while others will be processed by application 2 and then by application 1; eventually, all packets will be processed by all registered applications. For comparison, we also plot the performance of GASPP when packet scheduling is disabled (GASPP-nosched), and the performance of having multiple standalone applications running on the GPU and the CPU.

We see that the throughput for GASPP converges to the throughput of the most intensive application. When combining the first two applications, the throughput remains at 33.9 Gbit/s. When adding the L7F (x=3), performance degrades to 18.3 Gbit/s. L7F alone reaches about 20 Gbit/s (Figure 11(a)). When adding AES (x=4), performance drops to 8.5 Gbit/s, which is about 1.93× faster than GASPP-nosched. The achieved throughput when running multiple standalone GPU-based implementations is about 16.25× lower than GASPP, due to excessive data transfers.

8 Limitations

Typically, a GASPP developer will prefer to port functionality that is parallelizable, and thus benefit from the GPU execution model. However, there may be parts of data processing operations that do not necessarily fit well on the GPU. In particular, middlebox functionality with complex conditional processing and frequent branching may require extra effort.

The packet scheduling mechanisms described in §5.3 help accommodate such cases by forming groups of packets that will follow the same execution path and will not affect GPU execution. Still, (i) divergent workloads that perform quite lightweight processing (e.g., which process only a few bytes from each packet, such as the FW application), or (ii) workloads where it is not easy to know which packet will follow which execution path, may not be parallelized efficiently on top of GASPP. The reason is that in these cases the cost of grouping is much higher than the resulting benefits, while GASPP cannot predict if packet scheduling is worth the case at runtime. To overcome this, GASPP allows applications to selectively pass network packets and their metadata to the host CPU for further post-processing, as shown in Figure 1. As such, for workloads that are hard to build on top of GASPP, the correct way is to implement them by offloading them to the CPU. A limitation of this approach is that any subsequent processing that might be required also has to be carried out by the CPU, as the cost of transferring the data back to the GPU would be prohibitive.

Another limitation of the current GASPP implementation is its relatively high packet processing latency. Due to the batch processing nature of GPUs, GASPP may not be suitable for protocols with hard real-time per-packet processing constraints.

9 Related Work

Click [19] is a popular modular software router that successfully demonstrates the need and the importance of modularity in software routers. Several works focus on optimizing its performance [10, 11].

SwitchBlade [8] provides a model that allows packet processing modules to be swapped in and out of reconfigurable hardware without the need to resynthesize the hardware. Orphal [18] and ServerSwitch [17] provide a common API for proprietary switching hardware, and leverages the programmability of commodity Ethernet switching chips for packet forwarding. ServerSwitch also leverages the resources of the server CPU to provide extra programmability. In order to reduce costs and enable quick functionality updates, there is an ongoing trend of migrating to consolidated software running on commodity “middlebox” servers [11, 15, 22].

GPUs provide a substantial performance boost to many network-related workloads, including intrusion detection [13, 24, 26] cryptography [14], and IP routing [12]. Many recent works also deal with GPU resource management in the OS [16, 21]. GPUfs [23] enhances the API available to GPU code, allowing GPU software to access host files directly. Finally, software mechanisms for tackling irregularities in both control flows and memory references have been proposed [29].

10 Conclusion

We have presented the design, implementation, and evaluation of GASPP, a flexible, efficient, and high-performance framework for network traffic processing applications. GASPP explores the design space of combining the massively parallel architecture of GPUs with 10GbE network interfaces, and enables the easy integration of user-defined modules for execution at the corresponding L2–L7 network layers. GASPP has been implemented using solely commodity, inexpensive components, and our development experiences further show that GASPP is easy to program using the C/CUDA language. We have used our framework to develop representative traffic processing applications, including intrusion detection and prevention systems, packet encryption applications, and traffic classification tools.

As part of our future work, we plan to investigate further how to schedule module execution on the CPU, and how these executions will affect the overall performance of GASPP. We also plan to implement an opportunistic GPU offloading scheme, whereby packets with hard real-time processing constraints will be handled by the host CPU instead of the GPU to reduce latency.

Acknowledgments. We would like to thank our shepherd KyoungSoo Park and the anonymous reviewers for their valuable feedback. This work was supported by the General Secretariat for Research and Technology in Greece with a Research Excellence grant.

References

[1] `http://code.google.com/p/back40computing/wiki/RadixSorting`.

[2] `http://l7-filter.sourceforge.net/`.

[3] `http://www.caida.org/data/passive/passive_2011_dataset.xml`.

[4] AES-NI. `https://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf`.

[5] CUDA Programming Guide. `http://http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[6] Libnids library. `http://libnids.sourceforge.net/`.

[7] Snort IDS/IPS. `http://www.snort.org`.

[8] Anwer, M. B., Motiwala, M., Tariq, M. b., and Feamster, N. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In *Proceedings of the ACM SIGCOMM 2010 conference* (2010).

[9] Dharmapurikar, S., and Paxson, V. Robust TCP stream reassembly in the presence of adversaries. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14* (2005).

[10] Dobrescu, M., Egi, N., Argyraki, K., Chun, B.-G., Fall, K., Iannaccone, G., Knies, A., Manesh, M., and Ratnasamy, S. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (2009).

[11] Ghodsi, A., Sekar, V., Zaharia, M., and Stoica, I. Multi-resource fair queueing for packet processing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012).

[12] Han, S., Jang, K., Park, K., and Moon, S. PacketShader: A GPU-accelerated Software Router. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (August 2010).

[13] Jamshed, M. A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., and Park, K. Kargus: a highly-scalable software-based intrusion detection system. In *Proceedings of the 2012 ACM conference on Computer and Communications Security* (2012).

[14] Jang, K., Han, S., Han, S., Park, K., and Moon, S. SSLShader: Cheap SSL Acceleration with Commodity Processors. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation* (March 2011).

[15] Joseph, D., and Stoica, I. Modeling Middleboxes. *Network, IEEE 22*, 5 (2008), 20–25.

[16] Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. TimeGraph: GPU scheduling for real-time multi-tasking environments. In *Proceedings of the 2011 USENIX Annual Technical Conference* (2011).

[17] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. ServerSwitch: a programmable and high performance platform for data center networks. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation* (2011).

[18] Mogul, J. C., Yalag, P., Tourrilhes, J., Mcgeer, R., Banerjee, S., Connors, T., and Sharma, P. API Design Challenges for Open Router Platforms on Proprietary Hardware. In *Proceedings of the ACM Workshop on Hot Topics in Networks* (2008).

[19] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The Click modular router. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999).

[20] Rizzo, L. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX conference on USENIX Annual Technical Conference* (2012).

[21] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011).

[22] Sekar, V., Egi, N., Ratnasamy, S., Reiter, M., and Shi, G. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012).

[23] Silberstein, M., Ford, B., Keidar, I., and Witchel, E. GPUfs: integrating a file system with GPUs. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[24] Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P., and Ioannidis, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection* (2008).

[25] Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E. P., and Ioannidis, S. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection* (2009).

[26] Vasiliadis, G., Polychronakis, M., and Ioannidis, S. MIDeA: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and Communications Security* (2011).

[27] Vasiliadis, G., Polychronakis, M., and Ioannidis, S. Parallelization and characterization of pattern matching using GPUs. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization* (2011).

[28] Vutukuru, M., Balakrishnan, H., and Paxson, V. Efficient and Robust TCP Stream Normalization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (2008).

[29] Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of the sixteenth international conference on Architectural Support for Programming Languages and Operating Systems* (2011).

Panopticon: Reaping the Benefits of Incremental SDN Deployment in Enterprise Networks

Dan Levin[†] Marco Canini[⋆] Stefan Schmid[†‡] Fabian Schaffert[†] Anja Feldmann[†]
[†]*TU Berlin* [⋆]*Université catholique de Louvain* [‡]*Telekom Innovation Labs*

Abstract

The operational challenges posed in enterprise networks present an appealing opportunity for automated orchestration by way of Software-Defined Networking (SDN). The primary challenge to SDN adoption in the enterprise is the deployment problem: How to deploy and operate a network consisting of both legacy and SDN switches, while benefiting from simplified management and enhanced flexibility of SDN.

This paper presents the design and implementation of Panopticon, an architecture for operating networks that combine legacy and SDN switches. Panopticon exposes an abstraction of a logical SDN in a partially upgraded legacy network, where SDN benefits can extend over the entire network. We demonstrate the feasibility and evaluate the efficiency of our approach through both testbed experiments with hardware switches and through simulation on real enterprise campus network topologies entailing over 1500 devices. Our results suggest that when as few as 10% of distribution switches support SDN, most of an enterprise network can be operated as a single SDN while meeting key resource constraints.

1 Introduction

Software-Defined Networking (SDN) has the potential to provide a principled solution to both simplify management and enhance flexibility of the network. SDN is a paradigm that offers a programmatic, logically-centralized interface for specifying the intended network behavior. Through this interface, a software program acts as a network controller by configuring forwarding rules on switches and reacting to topology and traffic changes.

While commercial SDN deployment started within data-centers [19] and the WAN [11], the roots of today's SDN arguably go back to the policy management needs of enterprise networks [4, 5]. In this paper, we focus on mid to large enterprise networks, *i.e.*, those serving hundreds to thousands of users, whose infrastructure is physically located at a locally-confined site. We choose this environment due to its complexity as well as the practical benefits that SDN network orchestration promises.

Enterprises stand to *benefit from SDN on many different levels*, including: (*i*) network policy can be declared over high-level names and enforced dynamically at fine levels of granularity [4, 8, 22], (*ii*) policy can dictate the paths over which traffic is directed, facilitating middlebox enforcement [28] and enabling greater network visibility, (*iii*) policy properties can be verified for correctness [15, 16], and (*iv*) policy changes can be accomplished with strong consistency properties, eliminating the chances of transient policy violations [30].

Existing enterprises that wish to leverage SDN however, face the problem of how to deploy it. SDN is not a "drop-in" replacement for the existing network: SDN redefines the traditional, device-centric management interface and requires the presence of programmable switches in the data plane. Consequently, the migration to SDN creates new opportunities as well as notable challenges:

Realizing the benefits. In the enterprise, the benefits of SDN should be realized as of the first deployed switch. Consider the example of Google's software-defined WAN [11], which required years to fully deploy, only to achieve benefits after a complete overhaul of their switching hardware. For enterprises, it is undesirable, and we argue, unnecessary to completely overhaul the network infrastructure before realizing benefits from SDN. An earlier return on investment makes SDN more appealing for adoption.

Eliminating disruption while building confidence. Network operators must be able to incrementally deploy SDN technology in order to build confidence in its reliability and familiarity with its operation. Without such confidence, it is risky and undesirable to replace all production control protocols with an SDN control plane as a single "flag-day" event, even if existing deployed switches already support SDN programmability. To increase its chances for successful adoption, any network control technology, including SDN, should allow for a small initial investment in a deployment that can be gradually widened to encompass more and more of the network infrastructure and traffic.

Respecting budget and constraints. Rather than a green field, network upgrade starts with the existing deployment and is typically a staged process—budgets are constrained, and only a part of the network can be upgraded at a time.

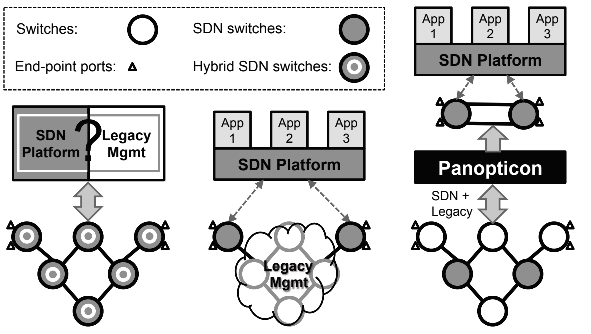
To address these challenges, we present **Panopticon**, a novel architecture for realizing an SDN control plane in a network that combines legacy switches and routers with SDN switches that can be incrementally deployed. We call such networks *transitional networks*. Panopticon abstracts the transitional network into a logical SDN, extending SDN capabilities potentially over the entire network. As an abstraction layer, Panopticon is responsible for hiding the legacy devices and acting as a "network hypervisor" that maps the logical SDN abstraction to the underlying hardware. In doing so, Panopticon overcomes key limitations of current approaches for transitional networks, which we now briefly review.

1.1 Current Transitional Networks

We begin with the "dual-stack" approach to transitional or "hybrid" SDN, shown in Figure 1a, where the flow-space is partitioned into several disjoint slices and traffic is assigned to either SDN or legacy processing [21]. To guarantee that an SDN policy applies to any arbitrary traffic source or destination in the network, the source or destination must reside at an SDN switch. Traffic within a flow-space not handled by SDN forwarding and traffic that never traverses an SDN switch may evade policy enforcement, making a single SDN policy difficult to realize over the entire network.

In summary, this mode's prime limitation is that it does not rigorously address how to realize the SDN control plane in a partial SDN deployment scenario, nor how to operate the resulting mixture of legacy and SDN devices as an SDN. It thus requires a contiguous deployment of hybrid programmable switches to ensure SDN policy compliance when arbitrary sources and destinations must be policy-enforced.

The second approach (Figure 1b) involves deploying SDN at the network access edge [6]. This mode has the benefit of enabling full control over the access policy and the introduction of new network functionality at the edge, *e.g.*, network virtualization [19]. Unlike a data-center environment where the network edge may terminate at the VM hypervisor, the enterprise network edge terminates at an access switch. At the edge of an enterprise network, to introduce new functionalities not accommodated by existing hardware involves replacing thousands of access switches. This mode of SDN deployment also limits the ability to apply policy to forwarding decisions within the network core (*e.g.*, load balancing, waypoint routing).



(a) Dual-stack (b) Access edge (c) Panopticon

Figure 1: Current transitional network approaches vs. Panopticon: (*a*) Dual-stack ignores legacy and SDN integration. (*b*) Full edge SDN deployment enables end-to-end control. (*c*) Panopticon partially-deployed SDN yields an interface that acts like a full SDN deployment.

1.2 Panopticon

Panopticon realizes an SDN control plane for incrementally deployable software-defined networks. Our main insight is that *the benefits of SDN to enterprise networks can be realized for every source-destination path that includes at least one SDN switch.* Thus, we do not mandate a full SDN switch deployment—a small subset of all switches may suffice. Conceptually, a single SDN switch traversed by each path is sufficient to enforce end-to-end network policy (*e.g.*, access control). Moreover, traffic which traverses two or more SDN switches may be controlled at finer levels of granularity enabling further customized forwarding (*e.g.*, traffic load-balancing).

Based on this insight, we devise a mechanism called the Solitary Confinement Tree (SCT), which uses VLANs to ensure that traffic destined to operator-selected switchports on legacy devices passes through at least one SDN switch. Combining mechanisms readily available in legacy switches, SCTs correspond to a spanning tree connecting each of these switchports to SDN switches, overcoming VLAN scalability limitations.

Just as many enterprise networks regularly divert traffic to traverse a VLAN gateway or a middlebox, a natural consequence of redirecting traffic to SDN switches is an increase in certain path lengths and link utilizations. As we discuss later (§4), deployment planning requires careful consideration to mind forwarding state capacities and to avoid introducing performance bottlenecks. Consequently, Panopticon presents operators with various resource-performance trade-offs, *e.g.*, between the size and fashion of the partial SDN deployment, and the consequences for the traffic.

As opposed to the dual-stack approach, Panopticon (Figure 1c) abstracts away the partial and heterogeneous deployment to yield a logical SDN. As we reason later (§ 2.4), many SDN control paradigms can be achieved in a logical SDN. Panopticon enables the expression of any end-to-end policy, as though the network were one big, virtual switch. Routing and path-level policy, *e.g.*, traffic engineering can be expressed too [2], however the abstract network view is reduced to just the deployed SDN switches. As more of the switches are upgraded to support SDN, more fine-grained path-level policies can be expressed.

In summary, we make the following contributions:

1. We design a network architecture for realizing an SDN control plane in a transitional network (§ 2), including a scalable mechanism for extending SDN capabilities to legacy devices.
2. We demonstrate the system-level feasibility of our approach with a prototype (§ 3).
3. We conduct a simulation-driven feasibility study and a traffic performance emulation study using real enterprise network topologies (with over 1500 switches) and traffic traces (§ 4).

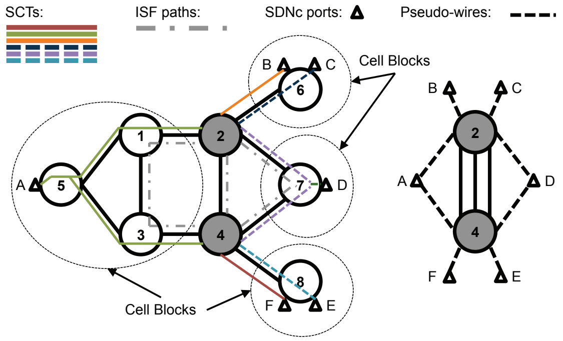
2 Panopticon SDN Architecture

This section presents the Panopticon architecture, which abstracts a transitional network, where not every switch supports SDN, into a logical SDN. The goal is to enable an SDN programming interface, for defining network policy, which can be extended beyond the SDN switches to ports on legacy switches as well.

Our architecture relies on certain assumptions underlying the operational objectives within enterprise networks. To verify these, we conducted five in-person interviews with operators from both large (≥10,000 users) and medium (≥500 users) enterprise networks and later, solicited 60 responses to open-answer survey questions from a wider audience of network operators [20].

Based on our discussions with network operators, and in conjunction with several design guidelines (*e.g.*, see [7, 13]), we make the following assumptions about mid to large enterprise networks and hardware capabilities. Enterprise network hardware consists primarily of Ethernet bridges, namely, switches that implement standard L2 mechanisms (*i.e.*, MAC-based learning and forwarding, and STP) and support VLAN (specifically, 802.1Q and per-VLAN STP). Routers or L3 switches are used as gateways to route between VLAN-isolated IP subnets. For our purposes, we assume a L3 switch is also capable of operating as a L2 switch. In addition, we assume that enterprises no longer intentionally operate "flood-only" hub devices for general packet forwarding.

Under these assumptions about legacy enterprise networks, Panopticon can realize a broad spectrum of logical SDNs: Panopticon can extend SDN capabilities to potentially every switchport in the network, however not every port need be included in the logical SDN. We envision an operator may conservatively choose to deploy Panopticon only in part of the network at first, to build confidence and reduce up-front capital expenditure, and then iteratively expand the deployment.



(a) Physical (b) Logical

Figure 2: Transitional network of 8 switches (SDN switches are shaded): (*a*) **The SCTs (Solitary Confinement Trees) of every SDNc (SDN-controlled) port overlaid on the physical topology.** (*b*) **Corresponding logical view of all SDNc ports, connected to SDN switches via pseudo-wires.**

To accommodate iterative expansion of the logical SDN, we divide the set of switchports in the network into *SDN-controlled (SDNc) ports*, that is, those that need to be exposed to and controlled through the logical SDN and *legacy ports*, those that are not. Note that while an SDNc port is conceptually an access port to the logical SDN network, it is not necessarily physically located on an SDN switch (see port *A* in Figure 2): It may be connected to an end-host or a legacy access switch.

We extend SDN capabilities to legacy switches by ensuring that all traffic to or from an SDNc port is always restricted to a *safe end-to-end path*, that is, a path that traverses at least one SDN switch. We call this key property of our architecture *Waypoint Enforcement*. The challenge to guaranteeing Waypoint Enforcement is that we may rely *only on existing mechanisms and features readily available* on legacy switches.

2.1 Realizing Waypoint Enforcement

Panopticon uses VLANs to restrict forwarding and guarantee Waypoint Enforcement, as these are ubiquitously available on legacy enterprise switches. To conceptually illustrate how, we first consider a straightforward, yet impractical scheme: For each pair of ports which includes at least one SDNc port, choose one SDN switch as the waypoint, and compute the (shortest) end-to-end path that includes the waypoint. Next, assign a unique VLAN ID to every end-to-end path and configure the legacy switches accordingly. This ensures that all forwarding decisions made by every legacy switch only send packets along safe paths. However, such a solution is infeasible, as VLAN ID space is limited to 4096 values, and often fewer are supported in hardware for simultaneous use. Such a rigid solution furthermore limits path diversity to the destination according and cripples fault tolerance.

Solitary Confinement Trees. To realize guaranteed Waypoint Enforcement in Panopticon, we introduce the concept of a *Solitary Confinement Tree* (SCT): a scalable Waypoint Enforcement mechanism that provides end-to-end path diversity. We first introduce the concepts of cell block and frontier. Intuitively, the role of a cell block is to divide the network into isolated islands where VLAN IDs can be reused. The border of a cell block consists of SDN switches and is henceforth called the *frontier*.

Definition 1 (Cell Blocks)**.** *Given a transitional network G,* Cell Blocks *$CB(G)$ is defined as the* set of connected components *of the network obtained after removing from G the SDN switches and their incident links.*

Definition 2 (Frontier)**.** *Given a cell block $c \in CB(G)$, we define the* Frontier *$\mathscr{F}(c)$ as the subset of SDN switches that are adjacent in G to a switch in c.*

Intuitively, the solitary confinement tree is a spanning tree within a cell block, *plus* its frontier. Each SCT provides a safe path from an SDNc port π to every SDN switch in its frontier—or if VLAN resources are scarce, a *subset* of its frontier, which we call the *active frontier*. A single VLAN ID can then be assigned to each SCT, which ensures traffic isolation, provides per-destination path diversity, and allows VLAN ID reuse across cell blocks. Formally, we define *SCT*s as:

Definition 3 (Solitary Confinement Tree)**.** *Let $c(\pi)$ be the cell block to which an SDNc port π belongs. And let* $\mathrm{ST}(c(\pi))$ *denote a spanning tree on $c(\pi)$. Then, the* Solitary Confinement Tree $\mathrm{SCT}(\pi)$ *is the network obtained by augmenting* $\mathrm{ST}(c(\pi))$ *with the (active) frontier $\mathscr{F}(c(\pi))$, together with all links in $c(\pi)$ connecting a switch $u \in \mathscr{F}(c(\pi))$ with a switch in* $\mathrm{SCT}(\pi)$.

Example. Let us consider the example transitional network of eight switches in Figure 2a. In this example, SCT (A) is the tree that consists of the paths $5 \rightarrow 1 \rightarrow 2$ and $5 \rightarrow 3 \rightarrow 4$. Instead note that SCT (B), which corresponds to the path $6 \rightarrow 2$, includes a single SDN switch because switch 2 is the only SDN switch adjacent to cell block $c(B)$. Figure 2b shows the corresponding logical view of the transitional network enabled by having SCTs. In this logical view, every SDNc port is connected to at least one frontier SDN switch via a pseudo-wire (realized by the SCT).

2.2 Packet Forwarding in Panopticon

We now illustrate Panopticon's basic forwarding behavior (Figure 3). As in any SDN, the control application is

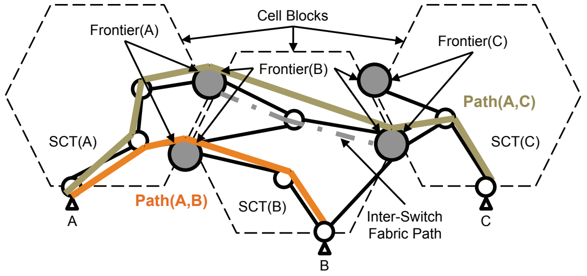


Figure 3: The forwarding path between A and B goes via the frontier shared by SCT (A) and SCT (B); the path between A and C goes via an Inter-Switch Fabric path connecting SCT (A) and SCT (C).

responsible for installing the necessary forwarding state at the SDN switches (*e.g.*, in accordance with the access policy) and for reacting to topology changes (fault tolerance is discussed in § 2.3).

Let us first consider traffic between a pair of SDNc ports s and t. When a packet from s enters $\mathrm{SCT}(s)$, the legacy switches forward the packet to the frontier based on MAC-learning, which establishes a symmetric path. Note that a packet from s may use a different path within $\mathrm{SCT}(s)$ to the frontier for each distinct destination. Once traffic toward t reaches its designated SDN switch $u \in \mathscr{F}(c(s))$, one of two cases arises:

SDN switches act as VLAN gateways. This is the case when the destination SDNc port t belongs to a cell block whose frontier $\mathscr{F}(c(t))$ shares at least one switch u with $\mathscr{F}(c(s))$. Switch u acts as the designated gateway between $\mathrm{SCT}(s)$ and $\mathrm{SCT}(t)$, that is, u rewrites the VLAN tag and places the traffic within $\mathrm{SCT}(t)$. For instance, in the example of Figure 2a, switch 2 acts as the gateway between ports A, B and C.

Inter-Switch Fabric (ISF). When no SDN switch is shared, we use an *Inter-Switch Fabric (ISF) path*: point-to-point tunnels between SDN switches which can be realized *e.g.*, with VLANs or GRE. In this case, the switch u chooses one of the available paths to forward the packet to an SDN switch $w \in \mathscr{F}(c(t))$, where w is the designated switch for the end-to-end path $p(s,t)$. In our example of Figure 2a, ISF paths are shown in gray and are used *e.g.*, for traffic from B or C to E or F, and vice versa.

We next turn to the forwarding behavior of legacy ports. Again, we distinguish two cases. First, when the path between two legacy ports only traverses the legacy network, forwarding is performed according to the traditional mechanisms and is unaffected by the partial SDN deployment. Policy enforcement and other operational objectives must be implemented through traditional means, *e.g.*, ACLs. In the second case, anytime a path between two legacy ports necessarily encounters an SDN switch, the programmatic forwarding rules at the switch can be leveraged to police the traffic. This is also the case for all traffic between any pair of an SDNc and a legacy port. In other words, Panopticon *always guarantees safe paths for packets from or to every SDNc port,* which we formally prove in the technical report [20].

2.3 Architecture Discussion

Having described all components of the architecture, we now discuss certain key properties.

Key SCT properties. Recall that one VLAN ID is used per SCT and that VLAN IDs can be reused across Cell Blocks. Limiting the size of the active frontier allows further VLAN ID reuse across fully-disjoint SCTs within the same cell block. A different path may be used within the SCT for each distinct destination. SCTs can be precomputed and automatically installed onto legacy switches (*e.g.*via SNMP) however, re-computation is required when the physical topology changes.

ISF path diversity trade-offs. Within the ISF, there may be multiple paths between any given pair of SDN switches. We expect that some applications may require a minimum number of paths. A minimum of two disjoint paths is necessary, to tolerate single link failures. If the ISF is realized using a VLAN-based approach, each path consumes a VLAN ID from every cell block it traverses. Alternative mechanisms, *e.g.*, IP encapsulation or network address translation can be used to implement the ISF depending on SDN and legacy hardware capabilities.

Coping with broadcast traffic. Broadcast traffic can be a scalability concern. We take advantage of the fact that each SCT limits the broadcast domain size, and we rely on SDN capabilities to enable in-network ARP and DHCP proxies as shown in [17]. We focus on these important bootstrapping protocols as it was empirically observed that broadcast traffic in enterprise networks is primarily contributed by ARP and DHCP [17, 26]. Last, we note that in the general case, if broadcast traffic must be supported, the overhead that Panopticon introduces is proportional to the number of SCTs in a cell block, which, at worst, grows linearly with the number of SDNc ports of a cell block.

Tolerating failures. We decompose fault tolerance into three orthogonal aspects. First, within an SCT, Panopticon relies on standard STP mechanisms to survive link failures, although to do so, there must exist sufficient physical link redundancy in the SCT. The greater the physical connectivity underlying the SCT, the higher the fault tolerance. Additionally, the coordination between SDN controller and legacy STP mechanisms allows for more flexible fail-over behavior than STP alone. When an SDN switch at the frontier $\mathscr{F}$ of an SCT notices an STP re-convergence, we can adapt the forwarding decisions at $\mathscr{F}$'s SDN switches to restore connectivity. A similar scheme can address link failures within the ISF.

Second, when SDN switches or their incident links fail, the SDN controller recomputes the forwarding state and installs the necessary flow table entries. Furthermore, precomputed fail-over behavior can be leveraged as of OpenFlow version 1.1 [29].

Third, the SDN control platform must be robust and available. In this respect, previous work [18] demonstrates that well-known distributed systems techniques can effectively achieve this goal.

2.4 Realizing SDN Benefits

By now, we have described how Panopticon shifts the active network management burden away from the legacy devices and onto the SDN control plane. This conceptually reduces the network to a logical SDN as presented in Figure 2b. Consequently, we want to be able to reason about what types of policy can be specified and which applications can be realized in such a transitional network.

Panopticon exposes an SDN abstraction of the underlying partial SDN deployment. In principle, any control application that runs on a full SDN can be supported in Panopticon since, from the perspective of the application, the network appears as though it is a full SDN deployment consisting of just the SDN switches. In practice, there are a small number of caveats.

SDNc ports in the logical SDN. An SDNc port in Panopticon is not necessarily physically located at an SDN switch, and it may be attached to multiple SDN switches. Accordingly, the SDN controller must take into account that each SDNc port may be reached from its frontier via multiple paths. Furthermore, visibility into how resources are shared on legacy links can not be guaranteed.

Logical SDN vs. full SDN. As an abstraction layer, Panopticon is responsible for hiding the legacy devices and acts as a "network hypervisor" that maps the logical SDN abstraction to the underlying hardware (similar to the concept of network objects in Pyretic [22]). However, because the global network view is reduced to the set of SDN switches, applications are limited to control the forwarding behavior based on the logical SDN. This should not be viewed strictly as a limitation, as it may be desirable to further abstract the entire network as a single virtual switch over which to define high-level policies (*e.g.*, access policy) and have the controller platform manage the placement of rules on physical switches [14]. Nevertheless, the transitional network stands to benefit in terms of management simplification and enhanced flexibility as we next illustrate.

More manageable networks. Arguably, as control over isolation and connectivity is crucial in the enterprise context we consider, the primary application of SDN is *policy enforcement*. As in Ethane [4], Panopticon enables operators to define a single network-wide policy, and the controller enforces it dynamically by allowing or preventing communication upon seeing the first packet of a flow as it tries to cross an SDN switch.

The big switch [14] abstraction enables the network to support Ethernet's plug-and-play semantics of flat addressing and, as such, simplifies the handling of host mobility. This can be observed from the fact that our architecture is an instance of the fabric abstraction [6]. The ISF represents the network core and SCTs realize the edge. At the boundary between the SCTs and ISF, the SDN switches enable the decoupling of the respective network layers, while ensuring scalability through efficient routing in the ISF.

More flexible networks. The controller maintains the global network view and performs route computation for permitted flows. This provides the opportunity to efficiently enforce middlebox-specific traffic steering within the SDN-based policy enforcement layer, as in SIMPLE [28]. Integrating middleboxes in Panopticon requires that middleboxes are connected to SDNc ports.

A logical SDN also enables the realization of strong consistency semantics for policy updates [30]. Although legacy switches do not participate in the consistent network update, at the same time, they do not themselves express network policy—as that forwarding state resides exclusively on SDN switches.

Putting it all together, Panopticon is the first architecture to realize an approach for operating a transitional network as though it were a fully deployed SDN, yielding benefits for the entire network, not just the devices that support SDN programmability.

3 Panopticon Prototype

To cross-check certain assumptions on which Panopticon is built, this section describes our implementation and experimental evaluation of a Panopticon prototype. The primary goal of our prototype is to demonstrate feasibility for legacy switch interaction–namely, the ability to leverage path diversity within each SCT, and respond to failure events and other behaviors within the SCT.

Our prototype is implemented upon the POX OpenFlow controller platform [1] and comprises two modules: path computation and legacy switch interaction.

Path computation. At the level of the logical SDN, our path computation module is straightforward: it reacts to the first packet of every flow and, if the flow is permitted, it uses the global network view to determine the shortest path to the destination. Consequently, it installs the corresponding forwarding rules. Our implementation supports two flow definitions: (1) the aggregate of packets between a pair of MAC addresses, and (2) the micro-flow, *i.e.*, IP 5-tuple. As each SDNc port may be reached over multiple paths from the SDN switches on its frontier, our prototype takes into account the behavior of STP within the SCT (monitored by the component below) to select the least-cost path based on source-destination MAC pair.

Legacy switch interaction. The Spanning Tree Protocol (STP) or a variant such as Rapid STP, is commonly used to achieve loop freedom within L2 domains and we interact with STP in two ways. First, within each SCT, we configure a per-VLAN spanning tree protocol (*e.g.*, Multiple STP) rooted at the switch hosting the SCT's SDNc port. We install forwarding rules at each SDN switch to redirect STP traffic to the controller, which interprets STP messages to learn the path cost between any switch on the frontier and the SCT's SDNc port, but *does not reply* with any STP messages. Collectively, this behavior guarantees that each SCT is loop free. When this component notices an STP re-convergence, it notifies the path computation module, which in turn adapts the forwarding decisions at SDN switches to restore connectivity as necessary. Second, to ensure network-wide loop freedom for traffic from legacy ports, SDN switches behave as ordinary STP participants. When supported, this is achieved by configuring STP on the switches themselves. Otherwise, Panopticon can run a functionally equivalent implementation of STP.

3.1 Application: Consistent Updates

To showcase the "logical SDN" programming interface exposed by Panopticon, we have implemented per-packet consistency [30] for transitional networks. Our application allows an operator to specify updates to the link state of the network, while ensuring that the safety property of per-packet consistency applies over the entire network, even to legacy switches.

To implement this application, we modify the path computation to assign a unique configuration version number to every shortest path between SDNc ports. This version number is used to classify packets according to either the current or the new configuration.

When the transition from current to new configuration begins, the controller starts updating all the SDN switches along the shortest path for both the forward and backward traffic. This update includes installing a new forwarding rule and using the IP *TOS* header field (i.e., in a monotonically increasing fashion) to encode or match the version number. The rules for the old configuration with the previous version number, if there are any, are left in place and intact. This procedure guarantees that any individual packet traversing the network sees only the "old" or "new" policy, but never both.

Once all the rules for the new configuration are in place at every switch, gratuitous ARP messages are sent over to the legacy switches along the new path so that the traffic is re-routed. After a operator-defined grace-period, when the last in-flight packet labeled with the "old" tag leaves the network, the controller deletes the old configuration rules from all the SDN switches, and the process completes.

3.2 Evaluation

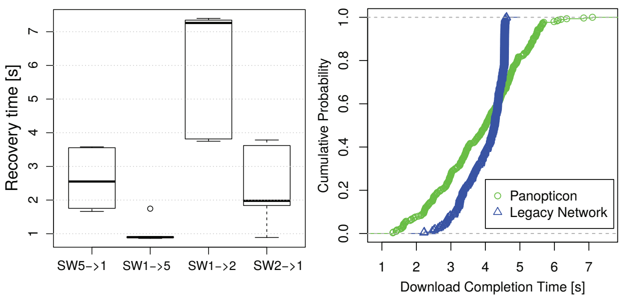
Our prototype is deployed on a network of hardware switches comprising two NEC IP8800 OpenFlow switches and one Cisco C3550XL, three Cisco C2960G, and two HP 5406zl MAC-learning Ethernet switches, interconnected as in Figure 2a. To emulate 6 hosts (*A* through *F*), we use an 8-core server with an 8-port 1Gbps Ethernet interface which connects to each SDNc port on the legacy switches depicted in the figure. Two remaining server ports connect to the OpenFlow switches for an out-of-band control channel.

We conduct a first experiment to demonstrate how Panopticon recovers from an STP re-convergence in an SCT, and adapts the network forwarding state accordingly. We systematically emulate 4 link failure scenarios between links (5,1) and (1,2) by disabling the respective source ports of each directed link. Host *A* initiates an iperf session over switch 2 to host *D*. After 10 seconds into the experiment, a link failure is induced, triggering an STP re-convergence. The resulting BDPU updates are observed by the controller and connectivity to host *D* is restored over switch 4. Figure 4a shows the elapsed time between the last received segment and first retransmitted packet over 10 repetitions and demonstrates how Panopticon quickly restores reachability after the failure event. Interestingly, we observe that Panopticon reacts faster to link changes detected via STP reconvergence (*e.g.*, `sw5_to_sw1`) than to link changes at the OpenFlow switches themselves (`sw1_to_sw2`), since our particular switches appear to briefly, internally delay sending those event notifications.

We next conduct a second experiment to explore how the SCT impacts the performance of a BitTorrent file transfer conducted among the hosts attached to SDNc ports. In this experiment, we begin by seeding a 100MB file at one host (*A* through *F*), in an iterative fashion. All other hosts are then initialized to begin simultaneously downloading the file from the seeder and amongst one another. We repeat each transfer 10 times, and measure the time for each host to complete the transfer. We then compare each time with an identical transfer in a L2 spanning tree topology. Figure 4b, illustrates that some of the hosts (*i.e.*, *A* and *D*) are able to leverage the multipath forwarding of their SCTs to finish sooner. Others, *e.g.*, *B* and *C* experience longer transfer times, as their traffic shares the same link to their frontier switch.

4 Incremental SDN Deployment

Panopticon makes no assumptions about the number of SDN switches or their locations in a partial SDN deployment. However, under practical resource constraints, an arbitrary deployment may make the *feasibility* of the logical SDN abstraction untenable, as the flow table capacities at the SDN switches and the availability of VLAN IDs on legacy switches are limited.



(a) SCT link failure recovery **(b) Bittorrent performance**

Figure 4: Testbed experiments: (*a*) **Panopticon recovers from link failure within seconds.** (*b*) **Panopticon enables path diversity but also increases load on some links.**

Beyond feasibility, the SDN deployment also influences network *performance*. By ensuring Waypoint Enforcement, SDN switches may become choke points that increase path lengths and link loads, in some cases beyond admissible values. Deployment planning therefore becomes a necessity.

4.1 Deployment Planning

Deciding the number and location of SDN switches to deploy can be viewed as an optimization problem wherein the objective is to yield a good trade-off between performance and costs subject to feasibility. We envision that a tool with configurable parameters and optimization algorithms may assist operators in planning the deployment by answering questions such as “What is the minimal number of SDN switches needed to support all ports as SDNc ports?” or “Which switches should be first upgraded to SDN to reduce bottleneck link loads?”

In a companion technical report of this paper [20], we present a general integer programming algorithm to compute a partial SDN deployment optimized for different objective functions and resource constraints. This algorithm can assist operators in upgrading the network, starting from a legacy network or one that is already partially upgraded.

We observe however that specific objectives and constraints for planning an SDN deployment are likely to depend on practical contextual factors such as hardware life-cycle management, support contracts and SLAs, long-term demand evolution and more. Unfortunately, these factors are rather qualitative, vary across environments, and are hard to generalize.

Instead, we reason more generally about how deployment choices influence feasibility and performance of our approach. To navigate the deployment problem space without the need to account for all contextual factors, we focus on a few general properties of desirable solutions:

(*i*) *Waypoint Enforcement:* Every path to or from an SDNc port must traverse at least one SDN switch.

Site	Access/Dist/Core	max/avg/min degree
LARGE	1296 / 412 / 3	53 / 2.58 / 1
EMULATED	489 / 77 / 1	30 / 6.3 / 1
MEDIUM	– / 54 / 3	19 / 1.05 / 1
SMALL	– / 14 / 2	15 / 3 / 2

Table 1: Evaluated network topology characteristics.

(ii) *Feasible:* SDN switches must have sufficient forwarding state to support all traffic policies they must enforce. VLAN requirements to realize SCTs must be within limits.

(iii) *Efficient:* The resulting traffic flow allocations should be efficient. We reason about efficiency using two metrics: The first metric is the path *stretch*, which we define for a given path (s,t) as the ratio between the length of the path under Waypoint Enforcement and the length of the shortest path in the underlying network. The second metric is the expected maximum load on any link.

4.2 Simulation-assisted Study

To explore feasibility and efficiency of Panopticon, we simulate different partial SDN deployment scenarios using real network topologies under different resource constraints and traffic conditions. These simulations let us (i) evaluate the feasibility space of our architecture, (ii) explore the extent to which SDN control extends to the entire network, and (iii) understand the impact of partial SDN deployment on link utilization and path stretch.

4.2.1 Methodology

To simulate Panopticon deployment, we first choose network topologies with associated traffic estimates and resource constraints.

Topologies. Detailed topological information, including device-level configurations, link capacities, and end-host placements is difficult to obtain for sizeable networks: operators are reluctant to share these details due to privacy concerns. Hence, we leverage several publicly available enterprise network topologies [34, 38] and the topology of a private, local large-scale campus network. The topologies range from SMALL, comprising just the enterprise network backbone, to a MEDIUM network with 54 distribution switches, to a comprehensive large-scale campus topology derived from anonymized device-level configurations of 1711 L2 and L3 switches. Summary information on the topologies is given in Table 1. Every link in each topology is annotated with its respective capacity. We treat port-channels (bundled links), as a single link of its aggregate capacity.

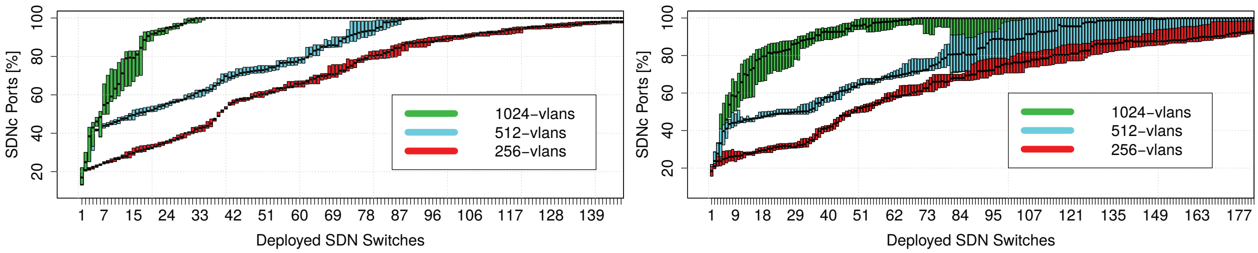
Simulation results on the SMALL and MEDIUM network gave us early confidence in our approach, however their limited size does not clearly demonstrate the most interesting design trade-offs. Thus, we only present simulation results for LARGE.

Focus on distribution switches. In our approach, we distinguish between *access switches*, *distribution switches*, and *core switches*. Access switches are identified both topologically, as well as from device-level configuration metadata. Core switches are identified as multi-chassis devices, running a L3 routing protocol. Due to their topological location, SCT construction to core switches becomes challenging, thus, we focus on distribution switches (in the following referred to as the *candidate set* for the upgrade). In case of the LARGE network, this candidate set has cardinality 412 of which, 95 devices are identified as L3 switches (running OSPF or EIGRP). Within this distribution network, we reason about legacy distribution-layer switchports as candidates to realize SDNc ports, subject to Waypoint Enforcement. Each distribution-layer switchport leads to an individual access-layer switch to which end-hosts are attached. Thus, we identify 1296 candidate SDNc ports. Unless otherwise noted, we construct SCTs connecting each SDNc port to its full frontier.

Traffic estimates. We use a methodology similar to that applied in SEATTLE [17] to generate a traffic matrix based on packet-level traces from an enterprise campus network, the *Lawrence Berkeley National Laboratory* (LBNL) [26]. The LBNL dataset contains more than 100 hours of anonymized packet level traces of activity of several thousands of internal hosts. The traces were collected by sampling all internal switchports periodically. We aggregate the recorded traffic according to source-destination pairs and for each sample, we estimate the load imposed on the network. We note that the data contains sources from 22 subnets.

To project the load onto our topologies, we use the subnet information from the traces to partition each of our topologies into subnets as well. Each of these subnets contains at least one distribution switch. In addition, we pick one node as the Internet gateway. We associate traffic from each subnet of the LBNL network in random round-robin fashion to candidate SDNc ports. All traffic within the LBNL network is aggregated to produce the intra-network traffic matrix. All destinations outside of the LBNL network are assumed to be reachable via the Internet gateway and thus mapped to the chosen gateway node. By running 10 different random port assignments for every set of parameters, we generate different traffic matrices, which we use in our simulations. Still, before using a traffic matrix we ensure that the topology is able to support it. For this purpose we project the load on the topology using shortest path routes, and scale it conservatively, such that the most utilized gigabit link is at 50% of its nominal link capacity.

Resource constraints. Although the maximum number of VLAN IDs expressible in 802.1Q is 4096, most mid- to high-end enterprise network switches support



(a) VOL Switch Selection Strategy

(b) Random Switch Selection Strategy

Figure 5: Percentage of SDNc ports as a function of deployed SDN switches, under different VLAN availability. When more VLANs are supported by legacy devices, more SDNc ports can be realized with fewer SDN switches.

512-1024 VLAN IDs for simultaneous use. Accordingly, we focus on simulating scenarios where legacy switches support at most 256, 512, and 1024 simultaneous VLAN IDs. While first generation OpenFlow capable switches were limited to around 1K flow table entries many current switches readily support from 10K to 100K entries for exact and wild-card matching. Bleeding edge devices support up to 1M flow table entries [25]. To narrow our parameter space, we fix the flow table capacity of our SDN switches to 100k entries, and vary the average number of rules required to realize policy for a single SDNc port from 10 to 20. We furthermore ensure that every SDN switch maintains at all times both policy and basic forwarding state (one entry per SDNc port reached through that switch) to ensure all-to-all reachability in the absence of any policy. We note, this is a conservative setting; by comparison, if flow table entries were kept only in the temporal presence of their respective, active source-destination traffic in the LBNL dataset, the maximum number of entries would never exceed 1,200 flows/s [4].

4.2.2 Switch Deployment Strategies

Given our topology and traffic estimates, we next explore how SDN switch deployment influences feasibility and performance. We study this through a simple yet effective deployment heuristic inspired by classical techniques such as Facility Location, called VOL.

VOL iteratively selects one legacy switch to be replaced at a time, in decreasing order of switch egress traffic volume. SDNc candidate ports are then accommodated in the following greedy fashion: SDNc ports from the previous iteration are accommodated first (we initially iterate over a random permutation of SDNc candidates). An SCT is constructed to the active frontier, whose size, chosen by the designer, defines a feasibility-efficiency trade-off we investigate later. If an SCT can be created, designated SDN switches from the active frontier are selected for each destination port, and flow table entries are allocated. If flow table policy is accommodated, the traffic matrix is consulted and traffic is projected from the candidate port to every destination along each waypoint-enforced path. When no link exceeds its maximum utilization (or safety threshold value), the port is considered SDNc. The remaining SDNc candidates are then tried and thereafter, the next SDN switch candidate is deployed and the process repeats. As VOL is a greedy algorithm and does not backtrack, it may terminate prior to satisfying all SDNc candidates, despite the existence of a feasible solution.

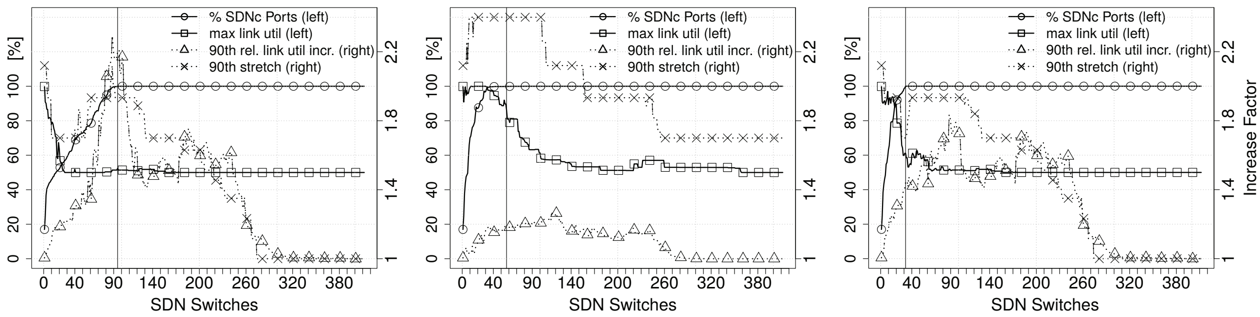
For comparison, we make use of **RAND**, which iteratively picks a legacy switch uniformly at random, subject to VLAN, flow table, and link utilization constraint satisfaction. RAND allows us to evaluate the sensitivity of the solution to the parameters we consider and the potential for sophisticated optimizations to outperform naïve approaches. We repeat every RAND experiment with 10 different random seeds.

4.2.3 SDNc Ports vs. Deployment Strategy

As Panopticon is designed to enable a broad spectrum of partial SDN deployments, we begin our evaluation by asking, “As a deployment grows, what fraction of candidate SDNc ports can be accommodated, under varying resource constraints?”

Scenario 1: To answer this question, we choose three values for the number of maximum simultaneous VLANs supported on any legacy switch (256, 512, 1024). We choose a policy requirement of 10 flow table entries on average for every (SDNc, destination port) pair as defined in the traffic matrix, so as to avoid a policy state bottleneck. We reason that policy state resource bottlenecks can be avoided by the operator by defining worst-case policy state needs in advance and then deploying SDN switches with suitable flow table capacity. We then compare our two deployment strategies VOL and RAND for different numbers of deployed SDN switches, as depicted by Figure 5 in which repeated experimental runs are aggregated into boxplots.

Observations 1: Figure 5 illustrates that the ability to accommodate more SDNc ports with a small number of SDN switches depends largely on the number of VLAN IDs supported for use by the legacy hardware. Under favorable conditions with 1024 VLANs, 100% SDNc port



(a) 512 VLANs, Full Frontier **(b) 512 VLANs, 2 Active Frontier** **(c) 1024 VLANs, Full Frontier**

Figure 6: SDN deployment vs. max link utilization, 90th percentile path stretch and relative link util increase. Feasible 100% SDNc port coverage can be realized with 33 SDN switches, with acceptable link utilization and path stretch.

coverage can be had for as few as 33 SDN switches. VLAN ID availability is necessary to construct SCTs and in Figure 5a we see that when legacy switches support at most 256 VLANs, over 140 SDN switches must be deployed before achieving full SDNc port coverage. Figure 5b shows the importance of choosing where to deploy SDN switches, as the earliest 100% SDNc feasible solution requires 20 additional SDN switch over VOL.

4.2.4 How Will Panopticon Affect Traffic?

We next ask: "As more SDNc ports are accommodated, what will Waypoint Enforcement do to the traffic?"

Scenario 2: To answer this question, we evaluate the metrics path stretch and link utilization as we increase the SDN deployment, subject to two different VLAN resource constraints. As in Scenario 1, we assume average policy requirement of 10 flow table entries for every (SDNc, destination port) pair. Recall that our methodology scales up the baseline traffic matrix to ensure that the most utilized link in the original network is 50% utilized.

Figure 6 plots the relationship between the percentage of accommodated SDNc ports, the maximum link utilization, and the 90th percentile link utilization path stretch. Median values are shown for all metrics, across the repeated experiments. The feasible regions of each full "logical SDN" deployment with respect to all resource constraints are indicated by the vertical bar.

Observations 2: Figure 6a indicates that with 512 VLANs usable in the legacy network, a full logical SDN becomes feasible with 95 switches where the most utilized link reaches 55% of its capacity. The 90th percentile path stretch hovers around 2.1. As further switches are upgraded, the stretch and relative link utilization continue to improve. A more optimistic case is depicted in Figure 6c where full logical SDN is achieved with 33 switches. However, given fewer SDN waypoints, the maximum link utilization is higher at 60%. The key takeaway from this plot is that given conservative base link utilization, the additional burden imposed by SDN Waypoint Enforcement is small in many deployments.

4.2.5 Efficient 100% SDNc Port Feasibility

As we point out in our architecture section, Panopticon allows the designer to make efficiency trade-offs, where a full logical SDN can be realized with fewer SDN switches, at the expense of higher link utilization and path stretch. The parameter that governs this trade-off is the active frontier size. We next look to Figures 6a and 6b, which illustrate how this trade-off plays out.

Recall from Figure 6a that for a legacy network supporting 512 VLANs, a full logical SDN becomes feasible with about 95 SDN switches when using all available frontier switches. However, each path to the frontier switches consumes a VLAN, which blocks other SDNc candidate ports later on. By limiting the active frontier to at most 2 switches, Figure 6b illustrates that a feasible solution can be achieved with 56 switches. The path stretch notably increases to a factor of 2.4, compared to less than 2 when a larger frontier is used. This trade-off underlines the flexibility of Panopticon: Operators can make design choices tailored to their individual network performance requirements.

4.3 Traffic Emulation Study

To compliment our simulation-based approach and further investigate the consequences of Panopticon on traffic, we conduct a series of emulation-based experiments on portions of a real enterprise network topology. These experiments (i) provide insights into the consequences of Waypoint Enforcement on TCP flow performance, and (ii) let us explore the extent to which the deployment size impacts TCP flow performance when every access port is operated as an SDNc port.

Setup. We use Mininet [10] to emulate a Panopticon deployment. Due to the challenges of emulating a large network [10], we scale down key aspects of the network characteristics of the emulation environment. We (i) use a smaller topology, EMULATED (see Table 1), which is a 567-node sub-graph of the LARGE topology obtained by pruning the graph along subnet boundaries, (ii) scale down the link capacities by 2 orders of magnitude, and

	min	median	avg	max
Flow Sizes (in MB)	0.00005	6.91	9.94	101.70
Path Stretch A	1.0	1.0	1.002	1.67
Path Stretch B	1.0	1.0	1.16	3.0
Path Stretch C	1.0	1.33	1.25	3.0

Table 2: Traffic parameter and path stretch statistics.

(iii) correspondingly reduce the TCP MSS to 536 bytes to reduce packet sizes in concert with the reduced link capacities. This allows us to avoid resource bottlenecks that otherwise interfere with traffic generation and packet forwarding, thus influencing measured TCP throughput.

We run our experiments on a 64-core at 2.6GHz AMD Opteron 6276 system with 512GB of RAM running the 3.5.0-45-generic #68 Ubuntu Linux kernel using Open-VSwitch version 2.1.90. Baseline throughput tests indicate that our system is capable of both generating and forwarding traffic of 489 simultaneous TCP connections in excess of 34Gbps, sufficiently saturating the aggregate emulated link capacity of every traffic sender in our experiments. We note that traffic in the subsequent experiments is generated on the system-under-test itself.

Thus, our emulation experiments involve 489 SDNc ports located at “access switches” at which traffic is sent into and received from the network. The distribution network consists of 77 devices of which 28 devices are identified as IP router gateways that partition the network in Ethernet broadcast domains. Within each broadcast domain, we introduce a single spanning tree to break forwarding loops.

Traffic. We apply a traffic workload to our emulated network based on (i) a traffic matrix, defined over the 489 SDNc ports, and (ii) a synthetically generated flow size distribution where individual TCP flow sizes are obtained from a Weibull distribution with shape and scaling factor of 1, given in Table 2.

We re-use the traffic matrix used in the simulations to define the set of communicating source-destination pairs of SDNc ports in the network. For system scalability reasons, we limit the number of source-destination pairs to 1024, selected randomly from the traffic matrix. For each pair of SDNc ports, we define a sequence of TCP connections to be established in iterative fashion, whose transfer sizes are determined by the aforementioned Weibull distribution. The total traffic volume exchanged between each pair is limited to 100MB. When the experiment begins, every source-destination pair, in parallel begins to iterate through its respective connection sequence. Once every traffic source has reached its 100MB limit, the experiment stops.

Scenarios. We consider three deployment scenarios in which we evaluate the effects of Panopticon on TCP traffic: Scenario *A* in which 28 switches out of the 77 distribution switches are operated as SDN switches, and scenarios *B* and *C*, which narrow down the number of SDN switches in *A* to 10 and 5 SDN switches, respectively. SDN switch locations are selected at random based on location of IP routers, identified from the topology dataset.

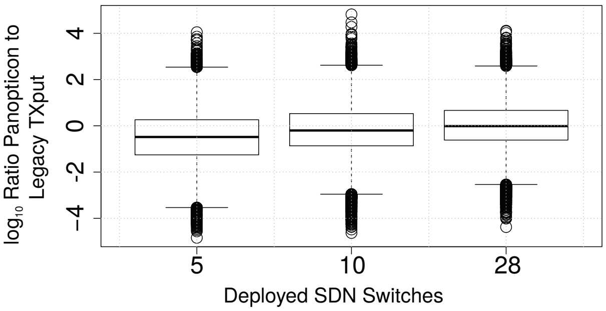


Figure 7: In both scenarios *A* and *B* (28 and 10 SDN switches), the median throughput over all experiments remains close to the performance of the legacy network.

Results. In each scenario, we compare TCP flow throughput in the Panopticon deployment versus the original network topology (which uses shortest-path IP routes with minimum cost spanning trees). Table 2 lists path stretch statistics for each scenario, namely, the ratio of SDN (waypoint-enforced) to legacy path length for every source-destination pair in the network.

Figure 7 illustrates the impact of Waypoint Enforcement on TCP performance in the three scenarios. The first observation we make is that in scenario *A*, when the 28 IP routers are replaced with SDN switches, the impact on median TCP throughput is negligible. This is perhaps expected, as all traffic across subnets must traverse some IP router in the legacy network, regardless. Some flows experience congestion due to Waypoint Enforcement. Other flows actually experience a performance increase due to the availability of multiple alternate paths in Panopticon. As the SDN deployment shrinks to more conservative sizes in scenarios *B* and *C*, the effects of Waypoint Enforcement becomes more prominent, supporting our observed simulation results.

4.4 Discussion

Scalability. As the number of SDNc candidates increases, the resource demands grow as well. We believe that one or two SDNc ports for every access switch however is a reasonable starting point for most partial SDN deployments. Even at one SDNc per access switch, a reasonable level of policy granularity, as end-hosts connected to the same physical access switch are often considered to be part of the same administrative unit as far as policy-specification is concerned. Should finer-grained SDNc port allocation be necessary, features such as Cisco’s protected switchports (or similar ones from other vendors) may be leveraged to extend Waypoint Enforcement to individual access-switch ports without the need for additional SCTs.

Why fully deploy SDN in enterprise networks? Perhaps many enterprise networks do not need to fully deploy SDN. As our results indicate, it is a question of the trade-offs between performance requirements and resource constraint satisfaction. Our Panopticon evaluation suggests that partial deployment may in-fact be the right mid-term approach for some enterprise networks.

5 Related Work

Our approach toward a scalable, incrementally deployable network architecture that integrates legacy and SDN switches to expose the abstraction of a logical SDN both builds upon and complements previous research.

SDN. In the enterprise, *SANE* [5] and *Ethane* [4] propose architectures to enforce centrally-defined, fine-grained network policy. Ethane overcomes SANE [5]'s deployment challenges by enabling legacy device compatibility. Ethane's integration with the existing deployment is however, ad-hoc and the behavior of legacy devices falls out of Ethane's control. Panopticon by contrast, can guarantee SDN policy enforcement through principled interaction with legacy devices to forward traffic along safe paths. Google's transition to a software-defined WAN involved an overhaul of their entire switching hardware to improve network performance [11]. In contrast to their goals, we take an explicit stance at transitioning to an SDN control plane without the need for a complete hardware upgrade. Considering a partial SDN deployment, Agarwal *et al.* [2] demonstrate effective traffic engineering of traffic that crosses at least one SDN switch. Panopticon is an architecture that enforces this condition for all SDNc ports. The work on software-controlled routing protocols [36] presents mechanisms to enable an SDN controller to indirectly program L3 routers by carefully crafting routing messages. We view this work as complementary to ours in that it could be useful to extend Waypoint Enforcement to IP routers.

Enterprise network design and architecture. Scalability issues in large enterprise networks are typically addressed by building a network out of several (V)LANs interconnected via L3 routers [7, 13]. *TRILL* [27] is an IETF Standard for so-called *RBridges* that combine bridges and routers. Although TRILL can be deployed incrementally, we are not aware of any work regarding its use for policy enforcement in enterprise networks.

Sun *et al.* [33] and Sung *et al.* [34] propose a systematic redesign of enterprise networks using parsimonious VLAN allocation to ensure reachability and provide isolation. These works focus on legacy networks only. The *SEATTLE* [17] network architecture uses a one-hop DHT host location lookup service to scale large enterprise Ethernet networks. However, such clean-slate approach is not applicable for the transitional networks we consider.

Scalable data-center network architectures. There is a wealth of recent work towards improving data-center network scalability. To name a few, *FatTree* [3], *VL2* [9], *PortLand* [24], *NetLord* [23], *PAST* [32] and *Jellyfish* [31], offer scalable alternatives to classic data-center architectures at lower costs. As clean-slate architectures, these approaches are less applicable to transitional enterprise networks, which exhibit less homogeneous structure and grow "organically" over time.

Evolvable inter-networking. The question of how to *evolve* or run a *transitional* network, predates SDN and has been discussed in many contexts, including Active Networks [37]. Generally, changes in the network layer typically pose a strain to network evolution, which lead to overlay approaches being pursued (*e.g.*, [12, 35]). In this sense, the concept of Waypoint Enforcement is grounded on previous experience.

6 Conclusion

SDN promises to ease network management through principled network orchestration. However, it is nearly impossible to fully upgrade an existing legacy network to an SDN in a single operation.

Accordingly, we have developed Panopticon, an enterprise network architecture realizing the benefits of a logical SDN control plane from a transitional network which combines legacy devices and SDN switches. Our evaluation highlights that our approach can deeply extend SDN capabilities into existing legacy networks. By upgrading between 30 to 40 of the hundreds of distribution switches in a large enterprise network, it is possible to realize the network as an SDN, without violating reasonable resource constraints. Our results motivate the argument, that partial SDN deployment may indeed be an appropriate mid-term operational strategy for enterprise networks. Our simulation source code is available at `http://panoptisim.badpacket.in`.

Acknowledgments. We wish to thank the network administrators and Rainer May for providing us insights into their operations. Thanks to Srinivas Narayana for giving us an early implementation of STP on POX. We thank Ho-Cheung Ng, Julius Bachnick and Arno Töll who contributed to an early Panopticon prototype. Thanks to Odej Kao and Bjoern Lohrmann for providing cluster time and support. We wish to thank Fred Baker, James Kempf, David Meyer, Jennifer Rexford, Srini Seetharaman, and Don Towsley as well as our reviewers for their valuable comments.

References

[1] POX Controller. `http://noxrepo.org`.

[2] S. Agarwal, M. Kodialam, and T. V. Lakshman. Traffic Engineering in Software Defined Networks. In *INFOCOM*, 2013.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[4] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, 2007.

[5] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security Symposium*, 2006.

[6] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: a Retrospective on Evolving SDN. In *HotSDN*, 2012.

[7] Cisco. Campus Network for High Availability Design Guide, 2008. `http://bit.ly/1ffWkzT`.

[8] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[9] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[10] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible Network Experiments Using Container-Based Emulation. In *CoNEXT*, 2012.

[11] S. Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[12] D. Joseph, J. Kannan, A. Kubota, K. Lakshminarayanan, I. Stoica, and K. Wehrle. OCALA: An Architecture for Supporting Legacy Applications over Overlays. In *NSDI*, 2006.

[13] Juniper. Campus Networks Reference Architecture, 2010. `http://juni.pr/1iR0vaZ`.

[14] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "One Big Switch" Abstraction in Software-Defined Networks. In *CoNEXT*, 2013.

[15] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking For Networks. In *NSDI*, 2012.

[16] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. In *NSDI*, 2013.

[17] C. Kim, M. Caesar, and J. Rexford. Floodless in SEATTLE: A Scalable Ethernet Architecture for Large Enterprises. In *SIGCOMM*, 2008.

[18] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[19] T. Koponen et al. Network Virtualization in Multi-tenant Datacenters. In *NSDI*, 2014.

[20] D. Levin, M. Canini, S. Schmid, and A. Feldmann. Panopticon: Reaping the benefits of partial sdn deployment in enterprise networks. Technical report, TU Berlin, 2013.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM CCR*, 38(2), 2008.

[22] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *NSDI*, 2013.

[23] J. Mudigonda, P. Yalagandula, J. Mogul, B. Stiekes, and Y. Pouffary. NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.

[24] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *SIGCOMM*, 2009.

[25] NoviFlow. NoviSwitch 1248 Datasheet. `http://bit.ly/1baQd0A`.

[26] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A First Look at Modern Enterprise Traffic. In *IMC*, 2005.

[27] R. Perlman, D. Eastlake, D. G. Dutt, S. Gai, and A. Ghanwani. Routing Bridges (RBridges): Rbridges: Base Protocol Specification IETF RFC 6325. 2009.

[28] Z. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *SIGCOMM*, 2013.

[29] M. Reitblatt, M. Canini, A. Guha, and N. Foster. FatTire: Declarative Fault Tolerance for Software-Defined Networks. In *HotSDN*, 2013.

[30] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. In *SIGCOMM*, 2012.

[31] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[32] B. Stephens, A. Cox, W. Felter, C. Dixon, and J. Carter. PAST: Scalable Ethernet for Data Centers. In *CoNEXT*, 2012.

[33] X. Sun, Y. E. Sung, S. D. Krothapalli, and S. G. Rao. A Systematic Approach for Evolving VLAN Designs. In *INFOCOM*, 2010.

[34] Y.-W. E. Sung, S. G. Rao, G. G. Xie, and D. A. Maltz. Towards Systematic Design of Enterprise Networks. In *CoNEXT*, 2008.

[35] N. Takahashi and J. M. Smith. Hybrid Hierarchical Overlay Routing (Hyho): Towards Minimal Overlay Dilation. *IEICE Transcations on Information and Systems*, E87-D(12), 2004.

[36] L. Vanbever and S. Vissicchio. Enabling SDN in Old School Networks with Software-Controlled Routing Protocols. In *Open Networking Summit (ONS)*, 2014.

[37] D. J. Wetherall. Service Introduction in an Active Network. *PhD Thesis, M.I.T.*, 1999.

[38] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. Automatic Test Packet Generation. In *CoNEXT*, 2012.

Programmatic Orchestration of WiFi Networks

Julius Schulz-Zander[†] , Lalith Suresh[†] , Nadi Sarrar[†] , Anja Feldmann[†] , Thomas Hühn[‡†] , and Ruben Merz[1]

TU Berlin, Germany[†] DAI-Labor, Germany[‡] Swisscom, Switzerland[1]

Abstract

With wireless technologies becoming prevalent at the last hop, today's network operators need to manage WiFi access networks in unison with their wired counterparts. However, the non-uniformity of feature sets in existing solutions and the lack of programmability makes this a challenging task. This paper proposes Odin, an SDN-based solution to bridge this gap. With Odin, we make the following contributions: *(i)* Light Virtual Access Points (LVAPs), a novel programming abstraction for addressing the IEEE 802.11 protocol stack complexity, *(ii)* a design and implementation for a software-defined WiFi network architecture based on LVAPs, and *(iii)* a prototype implementation on top of commodity access point hardware without modifications to the IEEE 802.11 client, making it practical for today's deployments. To highlight the effectiveness of the approach we demonstrate six WiFi network services on top of Odin including load-balancing, mobility management, jammer detection, automatic channel-selection, energy management, and guest policy enforcement. To further foster the development of our framework, the Odin prototype is made publicly available.

1 Introduction

Today's access networks are increasingly dominated by wireless technology at the last hop. Indeed, the WiFi Alliance, the certification authority for WiFi devices, reports almost 1.1 billion WiFi devices were shipped in 2011 [1], and predicts that this number will double by 2015. However, supporting this ever increasing number of wireless capable devices across residential, public, and enterprise networks is non-trivial and raises new challenges for network management, in particular for integrating wired, cellular, and wireless network management. To highlight this need, we point to the fact that large operators such as Deutsche Telekom (DT) [3] and Swisscom [6] are offloading data from their cellular networks to WiFi networks to reduce the stress on the former. Indeed, DT aims to deploy 2.5 million WiFi hotspots by 2016. Thus, these operators face the challenge of managing their different networks in unison and all the way to the users' premises.

Furthermore, modern enterprise WiFi networks typically consist of few dozens to thousands of Access Points (APs) serving a multitude of client devices including smart-phones, laptops, and tablets. For performance and scalability reasons, these networks require services which include mobility management, load-balancing, interference management, and channel reconfigurations. These services have to be realized as applications on top of the basic management functionality of the individual access points. However, different devices from different vendors typically offer different interfaces and do not offer native support for the needed applications. Additionally, today's enterprises and provider networks are Bring-Your-Own-Device (BYOD) networks, implying that the network has to accommodate an even more diverse set of user device types of different generations.

To manage this growing complexity, network operators need novel abstractions as well as new tools to uniformly manage the wired and wireless parts of their network, e.g., to verify network configurations, perform troubleshooting, or systematic debugging. In wired networks, recent advances in Software-Defined Networking (SDN) have enabled such features through programmatic control of networks. In an SDN, the control plane and data plane are decoupled, allowing network intelligence and state to be logically centralized. Using this centrally-available global view of the network, SDN allows operators to perform principled control and management of networks through the use of abstractions [26]. The best known SDN interface is OpenFlow, which specifies a protocol for a logically centralized controller to remotely manage forwarding tables within switches.

However, OpenFlow does not address the complexities of WiFi protocols and WiFi networks which include interference mitigation, mobility management, and channel selection techniques. This is unfortunate, because point-solutions exist for these WiFi-specific network problems but are often provided only by enterprise vendors through vertically integrated solutions. However, most cheap, off-the-shelf commodity hardware as deployed in today's access networks is outside the purview of such enterprise solutions.

Yet, proposals exist for extensible and programmable WiFi networks [20, 39]. However, these depend on client-side modifications which we argue is impractical to deploy. This is an obstacle not only for provider networks, but also for enterprise deployments given the trend towards BYOD.

We, in this paper, present Odin, an SDN-based solution which presents a programming abstraction which can provide the features enterprise and provider networks need. It bridges the gap between the range of features required by network operators and the lack of programmability in today's WiFi networks. In the process of designing Odin, we address the following research questions:

1. What programming abstractions are needed to address the complexities of the IEEE 802.11 protocol stack?
2. How can these abstractions be fit into an SDN architecture?
3. Can the SDN architecture already be realized on top of today's commodity access point hardware and without client modifications?

We find that the above questions can be answered affirmatively through the following contributions:

- The proposed Light Virtual Access Point (LVAP) abstraction captures the complexities of the IEEE 802.11 protocol stack.
- We present a prototype implementation of the LVAP approach which we have made publicly available[1].
- We evaluate the framework by presenting six typical WiFi network applications.

Odin is extensible in accordance with the features required in today's WiFi networks, whilst being deployable on top of low-cost commodity access point hardware. While we introduced the basic concept of LVAPs in our HotSDN workshop paper [30] and showed the system's capabilities in multiple demos [17, 24], this paper includes the detailed architecture for software-defined WiFi networks, a prototype implementation, as well as a system evaluation using multiple WiFi applications.

2 Use cases

Odin has been designed for the following use cases:

Traffic Offloading and Client Mobility: Offloading user's devices to WiFi allows operators to reduce stress on their cellular infrastructure. To this end, it is beneficial to provide users with consistent authentication credentials across their home networks, hotspots, and cellular connections, whilst managing client mobility. This will prevent the user from having to maintain multiple authentication credentials, whilst allowing operators to offload a user's traffic onto a hotspot when available. This is similar to what is proposed by the Hotspot 2.0 initiative, which however requires clients to support IEEE 802.11u. Furthermore, mobility management is an important feature within enterprise WiFi deployments, typically offered by today's vendors [5] and also explored by the research community [14, 18, 19, 20].

Network Performance Management: Channel selection, load balancing and wireless troubleshooting are crucial for the performance of WiFi networks, particularly within dense deployments like large enterprises or residential networks. Channel selection [7, 15, 35] involves continuously monitoring and then reacting to changes in the wireless environment. Load-balancing [9, 21] typically requires control of clients' attachment points to the network or the ability to hand off clients between WiFi access points. Lastly, there is a need for the ability to measure, detect, and localize interferers. This is because interference caused by non-WiFi devices can severely impact the achievable throughput of WiFi devices within the same vicinity [23], since both kinds of devices share the same wireless spectrum.

3 The Odin System

In this section, we describe the components of Odin and the Light Virtual Access Point (LVAP) abstraction.

3.1 Odin System Components

Figure 1 illustrates the components of the proposed design and their interactions. In line with the SDN concept, the design decouples the control from the data plane. This is done by having a logically centralized controller that leverages OpenFlow for the wired network, and a separate control plane protocol for the wireless part (elaborated upon in § 8). We chose to have separate protocols for programming the wired and wireless parts. This is because in its current state, OpenFlow does not extend well into the realm of the IEEE 802.11 MAC, as its scope is restricted to programming flow table rules on Ethernet- based switches. For instance, it cannot perform matching on wireless frames, cannot accommodate measurements of the wireless medium, report per-frame receiver side statistics, or be used for setting per-frame or -flow transmission settings for the WiFi datapath. We now describe the individual components in Odin:

[1] Odin source: `http://sdn.inet.tu-berlin.de`.

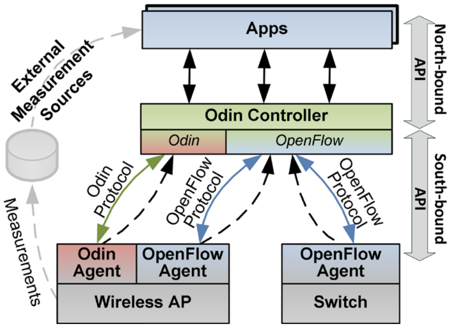


Figure 1: High-level design of the Odin architecture.

Odin controller: The controller enables network applications to programmatically orchestrate the underlying physical network. It exposes a set of interfaces to the applications (the northbound API) and then translates these calls into a set of commands on the network devices (the southbound API). The controller also maintains a view of the network including clients, APs, and OpenFlow switches, which the Odin applications can then control.

Odin agents: Agents run on the wireless APs and expose the necessary hooks for the controller (and thus applications) to orchestrate the WiFi network and report measurements. Time critical aspects of the WiFi MAC protocol (such as IEEE 802.11 acknowledgments) continue to be performed by the WiFi NIC's hardware. On the other hand, non time-critical functionality including management of client associations is implemented in software on the controller and the agents. This realizes a distributed WiFi split-MAC architecture. In addition, they perform matching on incoming frames to support a publish-subscribe system wherein network applications can subscribe to per-frame events.

Applications: For wireless network applications to take effective control decisions, they need access to statistics not only at a per- frame granularity, but also measurements of the medium itself (for instance, to infer interference from non-WiFi devices operating in the same spectrum). Thus, applications in Odin work either reactively or proactively by accessing measurements from multiple layers. This includes *(i)* measurements collected by the agents, *(ii)* OpenFlow statistics and *(iii)* measurements collected by external tools (e.g. `snmpd`). Odin applications can program the network through the northbound API offered by the controller.

3.2 Light Virtual Access Points

The Light Virtual Access Point (LVAP) is the abstraction in our system that allows us to address the specific requirements of WiFi networks, whilst allowing for unified management of the wired and wireless portions of the network. The LVAP is a per-client AP which simplifies the handling of client associations, authentication, handovers, and unified slicing of both the wired and wireless portions of the network. It enables a port-per-source view of WiFi networks akin to that of wired networks. In doing so, it remains orthogonal, but complementary, to trends in physical layer virtualization and RF spectrum slicing [29]. LVAPs are hosted on the agent, and their assignment to agents is controlled by the controller.

3.2.1 LVAPs as per-client APs

In regular IEEE 802.11 networks, clients need to associate with a physical AP before sending data frames. The association process begins with the discovery phase, where a client either actively scans for APs by generating probe requests, or passively learns about APs through beacon frames generated by the latter. During an active scan, APs that respond with probe response messages become candidates for the client to associate with. The client then decides which AP to associate with via a locally made choice. At this point, the association is defined between the client's MAC address and the BSSID of the AP. The BSSID of an AP is a MAC address of the AP's wireless interface and is different from the SSID, which is a network name.

This design of the WiFi protocol is inconvenient; there is no mechanism for centralized control over the client's association because the client makes the association decision entirely on its own. Furthermore, the infrastructure cannot instruct the client to re-associate without introducing additional signaling techniques such as [20].

The approach of LVAPs overcomes these difficulties without introducing additional signaling mechanisms between clients and the infrastructure, and thus conforms to our objective of not introducing client-side modifications. With LVAPs, every client receives a unique BSSID to connect to, essentially making them client-specific APs. Figure 2 indicates the decision flow in handling a client's association using LVAPs.

When a client probe scans, a new LVAP is spawned within the Odin agent on the physical AP. This LVAP then responds to the client with a probe response as instructed by the controller, following which, the clients completes the association handshake with its LVAP. As a result, a physical AP hosts a unique LVAP for each connected client. Every LVAP periodically unicasts beacon frames to its corresponding client. This ensures that a client never processes a beacon frame from another client's LVAP. The overhead of per-client beacon generation can be reduced by increasing the beacon interval, by setting the `NO_ACK` bit on the beacon frame, and also leveraging higher data-rates because of the unicast transmission. Note, beacons are typically broadcasted but are identical to probe response frames which are unicasted. Unicasting beacons does not confuse client devices (*cf.* 6.4).

As long as the client receives ACKs for the data frames it generates and receives beacons from the AP it is asso-

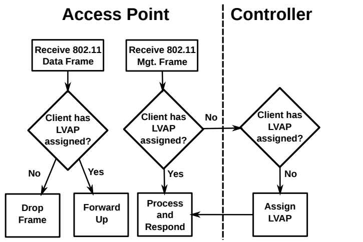

Access Point
Controller
Receive 802.11 Data Frame
Receive 802.11 Mgt. Frame
Client has LVAP assigned?
Client has LVAP assigned?
Client has LVAP assigned?
No
Yes
No
Yes
No
Drop Frame
Forward Up
Process and Respond
Assign LVAP

Figure 2: Processing path for WiFi frames: Agents invoke a controller for handling management frames.

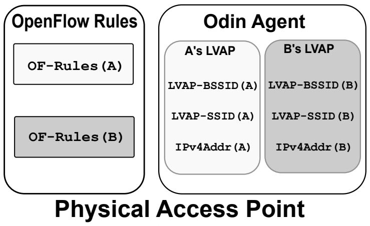


Figure 3: Per-Client LVAP state: a unique BSSID, set of SSIDs, client's IPv4 address, and OpenFlow rules (∼80 bytes of state excluding OF rules).

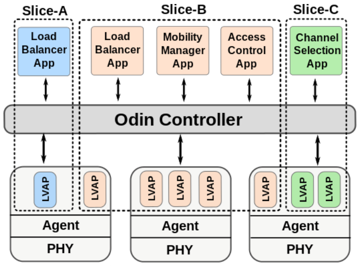


Figure 4: Odin applications operate upon a view of LVAPs and physical APs in their respective slices.

ciated to (in this case, an LVAP), the client stays associated. If the state corresponding to the client's LVAP is migrated to and instantiated at another Odin agent fast enough, the client does not attempt to re-scan (since from the client's point of view, its AP is still available). Thus, by migrating a client's LVAP between physical APs, the infrastructure can now control the client's attachment point to the network, without triggering a re-association at the client. The LVAP is thus an abstraction for the client's association state, and simplifies the expression of any handoff-based service like mobility managers and client load-balancers in the form of network applications. Since it does not introduce any additional signaling mechanism between the infrastructure and the client, it is legacy client compatible. In addition, it brings a port-per-source view of WiFi networks akin to that of wired networks, which simplifies fine-grained policy enforcement. Note, if a client experiences significant signal strength reduction as a result of an LVAP being migrated to a distant AP, the client will perform a regular re-scan.

While the notion of per-client BSSIDs is employed commercially to handle mobility [5], the concept of an LVAP is new. The LVAP as a programming abstraction solves problems that extend beyond mobility management, as we will demonstrate in this paper.

3.2.2 State Encapsulated by LVAPs

Figure 3 represents the state that is bound to each LVAP. For every associated client (identified by the client's WiFi MAC address), there is a corresponding LVAP which comprises the following information: a unique virtual BSSID, one or more SSIDs, the IP address of the client, and a set of OpenFlow rules. With encryption, the session key will be part of the LVAP state. When an LVAP is migrated from one physical AP to another, all corresponding state (the BSSID, SSIDs, IP address of the client, and OpenFlow rules) is migrated as well. Since the LVAP's BSSID is always consistent, the client does not perform a re-association. By binding a set of OpenFlow rules to the LVAP and allowing applications to program the wireless and wired side of the AP, we integrate our framework with OpenFlow.

3.2.3 Slicing and Control Logic Isolation with LVAPs

Accommodating multiple logical networks on top of the same physical infrastructure with different policies and control applications is called *network slicing*. A network slice is a virtual network with a specific set of SSIDs, where for example, the traffic may be VLAN tagged or directed to a specific destination port. Figure 4 indicates how slicing can be layered on top of LVAPs. A slice is defined as a set of physical APs (or agents), clients (and thus LVAPs), network applications, and one or more unique SSIDs. When clients attempt to associate to a particular SSID, they are automatically assigned to the slice to which the SSID belongs. Thus, the client and its LVAP are now assigned to the same slice. Applications operating on this slice can now manage the client (*e.g.*, perform migrations, or add/remove/update OpenFlow rules on the client's LVAP (*cf.* § 5)). The controller ensures that an application is only presented a view of the network corresponding to its slice. Since LVAPs are the primitive type upon which applications make control decisions, and applications do not have visibility of LVAPs from outside their slice, we thus achieve control logic isolation between slices.

3.2.4 Supporting Authentication Through LVAPs

Our architecture is compatible with the two most commonly deployed approaches for authentication.

WPA2 is the de-facto standard for authentication in today's WiFi networks (defined by IEEE 802.11i). In WPA2 Enterprise, a client authenticates against an authentication server with the AP acting as an authentication proxy to negotiate a session key. This session key is added to the client's LVAP state (*cf.* 3.2.2) and then used to encrypt the connection.

Guest WiFi: In this mode, a client's first HTTP request is redirected through OpenFlow rules associated with the LVAP to a login page. The authentication server returns a security token for the client to the controller after a successful authentication.

3.2.5 Multi-Channel Operation

Odin benefits from operating physical APs' wireless interfaces on the same channel for performing seamless client migration. However, when performing LVAP migrations between physical APs of different channels, the operation is similar to regular WiFi handovers where clients needs to perform a re-association. For multi-channel operation, Odin can leverage IEEE 802.11h (restricted to 5GHz band) to instruct clients to switch to a different channel while keeping association state intact. Additionally, Odin's port-per-source approach to managing clients with LVAPs is complementary to upcoming trends in RF spectrum slicing such as [29]. This will enable multiple LVAPs on the same AP to operate on different channels using a single antenna.

3.3 Reactive and Proactive Applications

Network applications written on top of Odin can function both reactively and/or proactively. Proactive applications are timer-driven whereas reactive applications use triggers and callbacks to handle events. The latter mode of operation is important particularly within WiFi networks due to the dynamic nature of the channel, and the system needs to react based on inputs from different measurement sources. To this end, in our current implementation, an application can utilize multiple measurement sources.

Measurements from the agent: Reactive applications make use of a publish-subscribe system of the Odin agent in order to have a handler invoked at the application whenever a per-frame event of interest occurs at the agents. In our current implementation, applications register thresholds for link-based (PHY and MAC layer) rx-statistics like receiver signal strength indicator (RSSI), bit-rate, and timestamp of the last received packet. For instance, an application can ask to be notified whenever a frame is received at an agent at an RSSI greater than -70dBm. In addition, applications can make use of measurements such as spectral scans that can be collected by the agents.

OpenFlow statistics: OpenFlow provides flow and port-based statistics of entries in switches' flow tables. Applications can query these statistics through the controller to make traffic-aware routing decisions.

External measurement sources: In addition to the usual per-link and per-flow statistics, applications can access data from multiple measurement sources outside the Odin framework, too, including the CPU and memory utilization and the channel active/busy times collected by tools such as `collectd`. We demonstrate this in § 5.

4 Odin on Commodity Hardware

In this section, we describe implementation details of the Odin prototype.

4.1 Controller

The controller is implemented as an extension to Floodlight OpenFlow controller. This allows us to use OpenFlow for Odin specific functionality such as tracking client IP addresses to be attached to their respective LVAPs by tapping into DHCP messages (*cf.* § 4.4). The initial assignment of agents to slices, the initial set of SSIDs per slice, and the network applications to run on each slice are defined via a configuration file. The controller uses a TCP-based control channel to invoke the Odin protocol commands on the agents (*cf.* 7). The controller organizes state on a per-slice basis, allowing it to present applications only a view of their respective slice in terms of associated clients, their LVAPs, and physical APs. Applications are expressed as Java code and run on top of the controller as threads. The programming API includes hooks for applications to view and control mappings of clients to APs, add/remove SSIDs to slices, and to register/unregister subscriptions for the pub-sub mechanism. As a result of using Floodlight, the controller is not distributed and runs on a single machine.

4.2 Agent

Odin agents run on physical APs, and are implemented in the Click Modular Router [16]. The agents implement the WiFi split-MAC together with the controller, host LVAPs, and collect statistics on a per-frame and host basis. They notify the controller whenever a frame is received that matches a per-frame event subscription registered by a particular application (*cf.* § 3.3). Alongside the agents, we run Open vSwitch on the APs to host OpenFlow rules carried by LVAPs as well as those expressed explicitly by network applications and the controller (for instance, to handle DHCP acknowledgments as described in § 3.2.2). Excluding the OpenFlow rules, the state associated with each LVAP hosted by an agent is approximately 48 bytes in size, and up to 32 bytes per-SSID in the slice (slices can announce multiple SSIDs).

4.3 ACK Generation

As mentioned in Section 3.2, the agent needs to ensure the IEEE 802.11 requirement of generating ACKs for each data frame that the client sends to its LVAP. ACK frame generation is handled in hardware by the WiFi cards due to their strict timing constraint. On Atheros WiFi cards, this is implemented using a BSSID mask register which indicates the common bits of all the BSSIDs being hosted on that card.

Whenever the card receives a valid frame, it verifies whether the destination address of the frame matches one of the BSSIDs it is hosting as per the bits set in the BSSID mask. If yes, an ACK frame is generated. However, a practical limitation exists with this mechanism. Consider the following two BSSIDs *02:00:00:00:00:02* and *02:00:00:00:00:01*. In this case, the last two bits are uncommon between the two BSSIDs, causing the mask to be *ff:ff:ff:ff:ff:fc*. This leads to the hardware ignoring the last two bits of the destination address of an incoming frame to decide whether to generate an ACK frame. In this case, a frame destined to *02:00:00:00:00:03* will also cause the hardware to generate an ACK, even though it is not hosting a BSSID with that value: a false positive.

In Odin, since we use one BSSID per client, this needs to be handled carefully. One way to overcome this issue is to assign BSSIDs to client LVAPs such that the mask on the AP where the LVAP is being assigned retains as many set bits as possible and remains orthogonal to the masks of neighboring APs. This can be achieved in software by the controller. Spreading LVAPs over multiple NICs and APs will also alleviate the problem. Another approach is to suppress spurious ACKs by modifying the check that the hardware performs upon receiving a frame. Today's low-end Broadcom WiFi cards support custom firmware such as OpenFWWF (our Atheros hardware does not support this). However, we conjecture that a programmable content-addressable memory for matching incoming frames in hardware enables possibilities beyond just selective ACK generation, with little increase in cost [2] and performance impact. This is particularly important as 802.11ac adoption is increasing, which supports throughputs on the order of 6.77 Gb/s. Recent work on software radios such as OpenRadio [8] will also aid in this direction.

4.4 LVAP Assignment

We now explain how Odin assigns LVAPs to clients.

Discovery: As per IEEE 802.11, clients perform active scans on all possible channels by broadcasting probe request messages. An agent that receives such a probe request forwards it to the controller. The controller then generates a BSSID unique to the client, and retrieves the list of SSIDs to announce (the union of SSIDs across all slices that the agent belongs to). It then instructs the agent to generate a probe response for each of these SSIDs, through the client-specific BSSID. This is how clients discover SSIDs being hosted via Odin.

Association: When a client tries to associate to a specific SSID, it generates probe requests that specify the corresponding SSID. An agent that receives such a probe request forwards the message to the controller. If the controller has not already created an LVAP for the client, it spawns an LVAP for the client on the agent from which this probe request was first received. The client is mapped to the slice that the SSID belongs to (an SSID can only be part of one slice). Once the LVAP is spawned for the client at an agent, the association is performed between the client and the LVAP. If a client does not associate to its LVAP within a configurable amount of time, it is removed from the agent. The agent process maintains a lookup table with the mappings of the client's MAC address to the LVAPs state (*cf.* 3.2.2). It then makes use of this per client state to prepare the right 802.11 frames and ARP packets when communicating with clients.

DHCP and ARP: The IP address of the client is required for the agent to correctly handle ARP requests that concern the client. The IP address of each client is obtained dynamically by the controller which sets up OpenFlow rules in order to receive an OpenFlow `PACKET_IN` event whenever a DHCP-ACK packet is received at an AP. This is done when an agent first registers with the controller. After a client associates and begins to obtain an IP address over DHCP, the controller receives the DHCP-ACK via OpenFlow, obtains the IP address, updates the client's LVAP, and then forwards the DHCP packet to the client via an OpenFlow `PACKET_OUT`.

5 Network Services on top of Odin

On top of our framework, we realized six different Odin applications which are correlated to the use cases described in § 2. For the evaluations, we use ten APs from our indoor testbed distributed across the 16^{th} floor (roughly 750 m^2) of the TEL building at the TU Berlin campus. The WiFi APs are based on embedded hardware (PC Engines Alix 3D2) equipped with Atheros IEEE 802.11abgn cards. All APs are running `OpenWrt` with the `ath9k` Linux driver, user-level Click, and Open vSwitch supporting OpenFlow version 1.0. The Odin controller runs on a x86-based server equipped with 2 CPUs at 2.1 GHz and 4 GB of RAM. We did not hit CPU or memory limitations in any of our experiments.

Application I: Mobility Manager

Supporting client mobility is a crucial feature in enterprise WiFi deployments. We have implemented a purely reactive mobility manager (89 source lines of code (SLOC)) on top of Odin, that leverages LVAP migrations. The application registers a subscription at the agents to be notified whenever an agent receives a frame at a receiver signal strength indicator (RSSI) above a specified value. Using context information passed through the corresponding callback (such as the exact value of the RSSI value and source that triggered the event), the application maintains a map of each client's RSSI value from the point of view of different agents. It then assigns the clients to the agents where they can get the best RSSI value, whilst subjecting its decisions to a hysteresis to prevent spurious oscillations of a client between APs. With legacy switches in the core, a packet is sent out by the new AP to trigger the "backwards learning" mechanism (ARP flushing) to setup new flow entries. With OpenFlow in the core, this can be achieved by updating flow entries along the new path.

We evaluate the architectural consequence of our reactive mobility manager's design, *i.e.*, the number of noti-

Table 1: Notifications generated between a handoff for two RSSI thresholds (T_{rss}) signal strength difference (Δ).

Frame Reception Rate (frames/sec)	$T_{rss} = -96\,dBm$		$T_{rss} = -76\,dBm$	
	$\Delta = 5$	$\Delta = 20$	$\Delta = 5$	$\Delta = 20$
1	13.2	15.8	13.0	16.2
1000	731.66	910.4	670.2	927.0
5000	3373.4	4609.2	3223.8	4515

fications required before performing a client handoff under a given mobility scenario. We show in § 6.2 that LVAP migrations have a negligible effect on the client's throughput. We note that this is only one example of a mobility manager that can be built atop Odin. As demonstrated in 5, Odin applications can utilize different metrics from multiple sources to base mobility decisions on.

Experiment scenario: We use two APs and a single x86-based client. The client associates to the network and initiates a UDP flow. We vary three parameters for the evaluation: (1) the threshold T_{rss} the application sets for subscription notifications, (2) the threshold Δ, *i.e.*, the minimum required difference of the client's RSSI observed at its current AP and potential new AP for the mobility manager to perform a handoff, and (3) the client's transmission rate. We artificially add a fixed offset to the client's RSSI value being recorded by the APs. Using this, we initially set the client's RSSI at the first AP to be 20dB more than at the other, and then reduce it by 0.1 unit every 100ms whilst increasing it at the other AP by the same amount. After 10s, the client's RSSI is higher at the second AP. When the difference is above Δ, the client is LVAP-migrated to the new AP. Thus, only the relative RSSI values of the client at the two APs affects the results (not the absolute values), which enables testing the application using a stationary client. We conduct 5 runs for each combination of parameters and average the results.

Results: Table 1 shows the results of our experiments for different combinations of T_{rss}, Δ, and the client's transmission rate. A decreasing T_{rss} leads to an increased number of notifications generated. A smaller Δ leads to the handoff being performed faster, and reduces the number of notifications in between handoffs. However, the dominant factor here is the transmission rate of the client itself. This shows that it is beneficial to introduce a rate-limiter for generating notifications by the agents. After all, for the same mobility scenario and during the handoff, there is a large number of notifications generated that do not further improve the mobility manager's decisions. Note, the framework cannot track clients that do not transmit any frames at all. One workaround is to use Odin's beacons as a mechanism to track idle clients at different physical APs. In regular WiFi, ACK frames do not contain the source address, but only the recipient address. Since beacons in Odin are unicast, they cause the client to generate an ACK frame addressed to their unique BSSID (which identifies the client). In order to reduce overhead, the system can set the `NO_ACK` bit on the beacons to avoid ACKs from active clients.

Application II: Load Balancer

The benefit of using a load-balancer in a WiFi setting is to increase the throughput for clients due to increased airtime fairness. To illustrate this, consider a scenario where there are multiple clients and one AP: each device gets almost the same share of channel access when operating at the same physical data rate. If only one of the clients generates upload traffic whereas the other stations only download data via the AP, the total upload throughput almost equals the combined throughput of the downloaders (since all download traffic is transmitted by the AP and it has to share channel access with a single uploader). This leads to airtime unfairness among the clients. With more APs and proper load balancing, this unfairness can be alleviated. Furthermore, load-balancing can lead to better resource utilization due to spacial reuse and the capture effect when the collision probability is high. The 802.11k amendment also attempts to address load-balancing, but requires modifications to the client.

Since LVAP-migrations are cheap, fast, and infrastructure-controlled (§ 6.2), client-migration based load-balancing is a good fit for an Odin application. We implemented a load-balancer (76 SLOC) to demonstrate the feasibility of such an application on top of Odin. This application queries the framework once per minute to obtain the list of clients that can be seen by different APs and their corresponding RSSI values. It uses this information to build a map of clients to lists of agents that are candidates for hosting their respective LVAPs. The application then evenly re-distributes LVAPs (clients) across physical APs, constrained by the hearing map.

Experiment scenario: We use up to ten APs. 32 clients automatically associate to the network and request files from a server. We use Harpoon [28] for flow-level traffic generation using a heavy-tailed flow size distribution, similar to traffic on the Internet. After the standard WiFi association, each client sends web requests to the Harpoon server. We conduct experiments with and without load-balancing enabled. Without load-balancing, the client is assigned to the first AP that receives the association request. With load-balancing, each LVAP is placed on a physical AP that has the highest RSSI and does not violate the client load on the AP. Because of the fixed PHY rate, no rate anomaly [31] can arise. We set the rate for management and data frames to the basic rate (6 Mbps). This ensures that all associated clients can exchange data with the APs.

Results: As expected, the overall TCP throughput increases when load-balancing is enabled (see Figure 5).

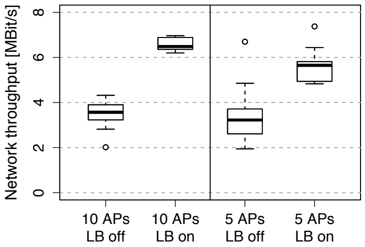


Figure 5: Throughput comparison with and without load-balancing.

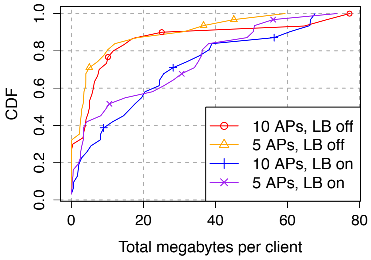


Figure 6: Total throughput per client with and without load-balancing.

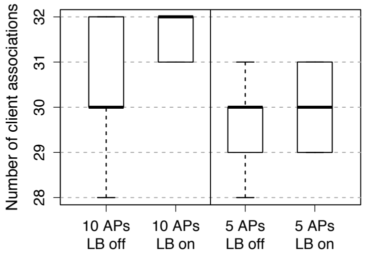


Figure 7: Number of clients contributing TCP traffic.

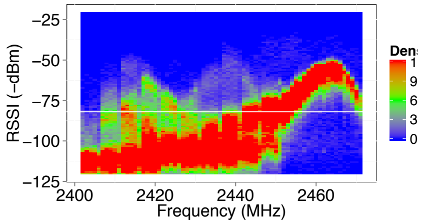


Figure 8: Troubleshooting detects jamming on frequency 2462MHz.

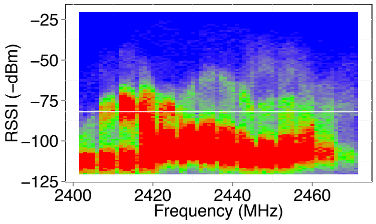


Figure 9: Spectral scan results during normal office working hours.

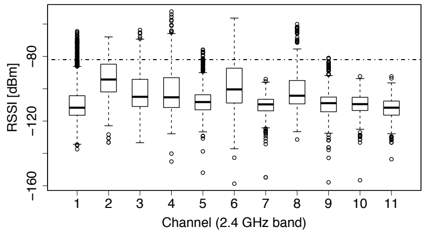


Figure 10: Snapshot of the 2.4 GHz band as seen by the ACS application.

Furthermore, the total throughput is increased when increasing the number of APs. The gain in throughput is attributed to spacial reuse and the capture effect when collisions occur. We observe that TCP connections were established by at least 28 clients across all runs with a median of 30 clients requesting data from the server (see Figure 7). Figure 6 shows the CDF of the per-client throughput of a single run. We observe an increase in fairness among clients with load balancing enabled: *i.e.*, roughly 50% of the clients were able to transmit around 20 MB of data with load balancing enabled compared to 15% without load-balancing. The gain of per-client throughput can be attributed to the previously mentioned spacial reuse, capture effect, and medium access probability of the APs, where each client gets roughly an equal share of airtime at the AP.

Application III: Wireless troubleshooting

Interference from non-WiFi devices such as microwave ovens, cordless phones, wireless security systems, and RF jammers can significantly impede the achievable throughput of nearby WiFi devices. To address this, interference identification systems (*e.g.*, Cisco CleanAir) are starting to become a part of today's enterprise deployments. These systems detect, localize, and quantify the interference impact caused by non-WiFi sources.

To this end, Odin leverages functionality of modern WiFi cards like Atheros AR9280 that provide coarse-grained energy samples per sub-carrier (frequency spacing of 312.5 KHz) of a WiFi channel. This provides the necessary interface for the development of systems like WiFiNet [23] on top of Odin for detection, localization and quantification of interference from a variety of non-WiFi interference sources.

Our troubleshooting application (102 SLOC) periodically (roughly every 5*s*) collects channel snapshots. Figure 8 shows the effect of a jammer (continuous stream of garbage frames) on channel 11 at 2462Mhz over a period of 5 minutes. This data can be used by a jammer detection application, e.g., to localize a jammer via triangulation.

Application IV: Automatic Channel Selection

Automatic Channel Selection (ACS) algorithms aim at automatically determining the best available channel for a WiFi interface. However, identifying combinations of channels for different APs while minimizing interference is challenging. Due the increasing amount of different channel bandwidths within the 2.4 and 5 GHz band. On top of Odin, an ACS application can query different channel properties from the agent (or external sources) for data that characterizes the channel properties. This includes, but is not limited to, spectral samples from the sub-carriers or the active- and busy-time in order to estimate the amount of interference on the channel.

We implemented a simple ACS (97 SLOC) application on top of Odin that is based on a per-AP channel selection scheme. It scans across all available channels and computes the average and the max RSSI for each channel center frequency. Based on multiple subsequent spectral scans, the ACS application picks the channel with the smallest maximum and average RSSI. This example Odin application can be extended to also utilize additional channel properties provided by the Odin agent or external data sources in order to estimate the channel load, e.g., channel active- and busy-time. This information can then be used to implement functionality akin to [25].

Figure 10 shows a snapshot of channel load of all center frequencies within the 2.4 GHz band during the day in our office environment.These snapshots are aggregated by our ACS application over time in order to get to a view similar to the one in Figure 9. Based on this aggregated view, the application then performs channel selection according to the heuristic described above. As indicated within the snapshot and the aggregated view, it can be seen that channel 11 is less utilized than channel 1, which we confirmed to be correlated with the number of APs operating on each channel.

Application V: Energy Efficient WiFi Networks
The problem of energy consumption in telecommunications infrastructure and mechanisms to address it have been studied in detail [12, 22, 33]. In earlier work [24] we demonstrated a system that uses Odin and leverages an integrated energy and mobility management system. The APs are organized into clusters, with each cluster having a master AP and multiple slave APs. The master APs always remain online and provide full coverage. Using a combination of observed network demand and an energy saving policy, the system activates or deactivates slave APs, and offloads clients between the master and the slaves accordingly. This is expressed as an energy manager written as an Odin application, which collects energy measurements via energy meters in order to make informed handover decisions.

Application VI: Guest policy enforcement
Centralized policy enforcement is an important requirement in enterprise WiFi deployments. This is one avenue where LVAPs complement OpenFlow-based access control particularly well. A guest network application uses the framework's API in order to instantiate a guest network on top of a slice of physical APs. It then attaches OpenFlow rules to all LVAPs of that slice which restricts the corresponding clients to be able to access only a certain set of subnets and ports. Since the OpenFlow entries follow the LVAP, other applications such as a mobility manager or load-balancer can operate on the same slice and perform LVAP migrations as well.

6 System Evaluation

In this section, we evaluate the CPU and memory utilization of the Odin controller as well as the latency involved in handling probe requests.

6.1 Controller load due to Pub-Sub

We evaluate the controller's CPU and memory utilization when running the mobility manager (*cf.* Section 5) under synthetically generated load. The aim is to understand the load involved in running a realistic application that makes use of the publish-subscribe subsystem.

We use nine APs of our testbed. The mobility manager is notified whenever a frame is received by any of the APs above a given signal strength threshold. Based on these notifications, the mobility manager decides on whether or not to trigger a client handover. A load generator running on a dedicated server invokes RPCs on the agents in order to mock client associations from a fixed list of clients. It then creates 1000 mock frame receptions per client per second at the APs at varying signal strengths to simulate the reception of arbitrary 802.11 frames. Depending on the signal strength of each frame, the agents notify the controller. Across different runs of the experiments we vary the number of clients as well as the number of APs that can overhear a single frame transmission by a client (*density factor*). The density factor determines how many APs generate a notification for a single frame transmission by a client. Each run of our load generator for a particular parameter takes 250 seconds. We repeat the experiment 10 times for each combination of the parameters and observe the steady state CPU and memory utilization.

We find that an increase in the number of clients for a fixed density factor leads to an increase in the controller's CPU utilization (see Figure 11). Furthermore, for a fixed number of clients, an increase in the density factor leads to an increased number of the mobility manager's subscriptions being triggered, leading to more control messages to the controller. For 500 clients with density factors of 5 and 7, our APs were CPU bottlenecked before being able to saturate the controller. However, we note that 500 is already a very large number of clients to support with only 9 APs. The memory utilization at the controller is 180 ± 7MB across all runs.

6.2 LVAP Handoff Micro-Benchmark

Since LVAPs are a central primitive of Odin, we perform experiments to gauge their effectiveness. The goal is to understand what performance related assumptions Odin applications can make. To this end, we compare LVAP-handoffs against standard WiFi handoffs. We also demonstrate that frequent LVAP-based handoffs do not affect the throughput of a TCP connection.

We use a single client and two APs of our testbed. An HTTP server in the same network acts as a traffic end-point. Since DHCP and authentication related delays only appear in the first connection to the network, the client is provided a static IP and no authentication is performed. Note that an LVAP handoff is not susceptible to the authentication delay. We conduct this experiment on a 5 GHz channel during the night to limit interference.

Comparison of Handoffs
For comparing the impact of handoffs, a client associates to an AP and begins an `HTTP` download of a large file. After 13 seconds, the client is made to handoff to another AP. When using Odin, the handoff uses an LVAP migration, whereas with regular WiFi, the client is explicitly told to perform a handoff using the `iw` command.

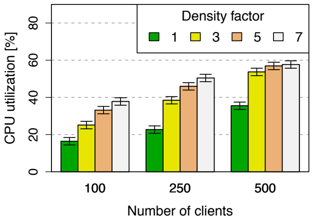


Figure 11: Normalized CPU utilization (2-cores) on the controller per density factor.

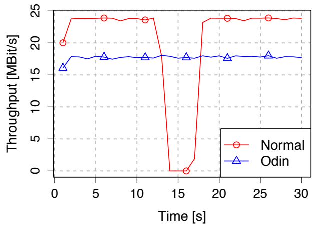


Figure 12: Impact of a LVAP migration and WiFi handoff on TCP throughput.

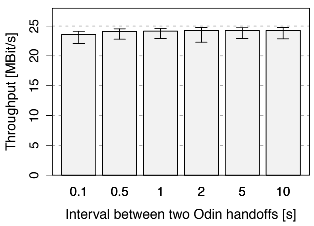


Figure 13: TCP throughput during Odin LVAP-handoffs.

Table 2: Latency for serving probe requests (excluding transmission time on the channel) across 9 APs

Scans per AP/s	Avg. Latency [ms]	Std-deviations [ms]
10	1.791	1.078
20	1.633	0.911
100	1.442	3.266
200	7.373	28.881

Figure 12 shows the TCP throughput over time with standard WiFi compared to Odin. For regular WiFi, the throughput drops to zero for several seconds before recovering. With Odin's LVAP handoff, the TCP throughput is unaffected. As Figure 12 indicates, there is an overall reduction of throughput (close to 5 Mbit/sec) with Odin as opposed to regular WiFi. This is, because we currently use userspace Click to run the Odin agents, resulting in slower and jittery forwarding performance on our APs which makes TCP to throttle down. However, this is orthogonal to continuously maintaining L2 and L3 connectivity, which Odin successfully achieves through LVAP migrations.

LVAP-Handoff frequency benchmark

To understand how often an LVAP-handoff can be executed against a client without affecting its performance, a single iperf-based TCP flow is executed with the client as the source over a period of 30 seconds. Between the 5^{th} and 25^{th} seconds of the measurement, LVAP-handoffs are repeatedly triggered between the two APs at fixed rates. Figure 13 shows that LVAP-based handoffs are leading to no significant throughput degradation of the TCP flow. Specifically, even when repeatedly performing LVAP-handoffs every 100 ms the throughput degradation is negligible. This illustrates the inexpensive nature of this operation. Furthermore, in the event of LVAP oscillations due to poorly written control-logic, client performance will not be impacted significantly.

6.3 Probe request serving latency

Since Odin invokes the controller for handling active-scans by clients, we evaluate whether our system can deliver probe responses to clients within the stipulated 30ms constraint.

For the experiment, a load-generator uses a hook on the agent that triggers the effect of a probe request reception. Nine APs of our testbed are used. We increase the rate of probe requests received at each agent. Each agent measures the time it takes in between receiving the probe request, informing the controller, having the controller respond with a BSSID, and then for the agent to construct a probe response message.

Table 2 shows, that the delays introduced due to our split-MAC design are well within the 30ms bound described above. We note that the latency is dominated by the network round-trip delay. Running the load-generator at 1,800 scans per-second (200 scans per-second-per-AP) lead to excessive queuing in the 100 Mbit/s Ethernet switch that our APs were connected to, which lead to the larger delays.

6.4 Compatibility with clients

We have tested our framework with common WiFi client devices, such as Windows, Linux, Mac OS X, iOS and Android devices. Compatibility with a multitude of client devices was demonstrated at [24, 17].

7 Related Work

We next position our work with respect to existing approaches that introduce programmability and/or perform centralized management of wireless networks.

Why not OpenFlow?: There have been efforts to bring OpenFlow to wireless APs (*e.g.*, using OpenFlow together with SNMP [38]). However, we argue that OpenFlow in its current state is ill-suited to orchestrate WiFi networks for many reasons. It cannot perform matching on wireless frames, cannot accommodate measurements of the wireless medium, report per-frame receiver side statistics, or be used for setting per-frame or -flow transmission settings for the WiFi datapath. Yet, extending OpenFlow to accommodate these requirements does not yield any specific benefits. By implementing a custom protocol for handling Odin agents, we thus achieve a cleaner separation of concerns.

Vendor solutions: A plethora of commercial enterprise WiFi solutions exist. These solutions typically manage APs centrally via a controller which is hosted either in the local network [5], or remotely in the cloud [4]. Unfortunately, these solutions do not extend into the purview of cheap low-cost commodity AP hardware that is used by provider networks, nor do they support common, open and programmable interfaces.

Virtual APs: Virtualization of APs have been studied in different contexts. [5] uses a one-BSSID-per-client approach to provide seamless mobility. SplitAP [10] pools together multiple APs in order to regulate air-time fairness. On the other hand, we demonstrate multiple use-cases for the LVAP abstraction as well as its utility as an API for building an SDN for WiFi networks.

Programmable wireless networks and centralized scheduling: Dyson [20] addresses the problem of extensibility in wireless LANs, by defining a set of APIs for clients and APs to be managed by a controller. The controller can query these nodes for channel information, form a global view of the network, and then control the network's behavior to enforce a set of policies. Flashback [11] proposes a control channel technique for WiFi networks, by allowing stations to send short control messages concurrently with data transmissions, without affecting throughput. This ensures a low overhead control plane for WiFi networks that is decoupled from the data plane. DIRAC [39] proposes a split-architecture wherein link-layer information is relayed by agents running on the APs to a central controller to improve network management decisions. However, these systems require special software or hardware on the client, which raises questions of practicality, and goes against the design requirements for our framework. There are systems that do not modify the client in order to deliver services. In DenseAP [19], channel assignment and association related decisions are made centrally by taking advantage of a global view of the network. However, it does not offer slicing of the WiFi, and provides a limited form of client association management because explicitly forces clients to disconnect, and then perform a re-scan in order to change the client's attachment point to the network. Thus, they do not perform client handoffs seamlessly.

CENTAUR [34] improves the data path in enterprise WiFi networks by using centralization to mitigate hidden terminals and to exploit exposed terminals. It is a natural fit for an application on top of Odin. FlowVisor [27] slices the network resources at the flow level and delegates control of different slices to controllers for wired networks. It achieves this by acting as a transparent proxy between OpenFlow switches and multiple OpenFlow controllers. This results in isolation of slices by ensuring that a controller operating on one slice cannot control traffic of another slice. With our framework, we have brought these concepts of isolation into WiFi networks. [37] supports multiple concurrently running experiments using slicing by SSIDs. However, as we show in this paper, slicing by BSSIDs as is done in Odin offers more powerful client isolation and management abilities.

8 Discussion and Further Steps

In designing Odin, we were careful to keep in mind upcoming trends in physical layer virtualization techniques, datapath programmability, hardware-based packet matching and operator requirements.

Virtualization of the PHY layer: Although we have addressed isolation at the IEEE 802.11 MAC layer, our system does not handle virtualization of the PHY layer, which is a logical next step. The IEEE 802.11 standard defines a Point Coordination Function (PCF), for centrally scheduled channel access. However, the PCF is rarely implemented in today's WiFi hardware/drivers. Picasso [29] enables virtualization across the MAC/PHY. It proposes a technique to perform spectrum slicing and allows a single radio to receive and transmit on different frequencies simultaneously. MAClets [13] allows multiple MAC/PHY protocols to share a single RF frontend. These advances can be used by Odin to operate multiple LVAPs with different characteristics on different channels on top of the same AP. Alternative approaches, such as [32] and [36], are incompatible with today's WiFi MAC/PHY and thus do not fit our design requirements.

Programmability of the WiFi data path: Odin's current implementation does not yet provide programmability of per-flow WiFi PHY settings. This is well within the scope of our design because the per-flow and -client transmission settings can be added as LVAP state. Enabling per-flow transmission settings will allow applications to centrally implement rate and power control. With OpenRadio [8], our system could also benefit from a clean-slate programmable network dataplane. This would allow Odin to work around hardware limitations such as that with the BSSID registers used for ACK generation (*cf.* § 4.3). We see OpenRadio, combined with Odin, as a steps towards WiFi networks that are fully programmable down to the PHY.

Performance isolation between slices: Odin in its current form achieves control logic isolation between slices. As of now, it is difficult to enforce FlowVisor-like bandwidth and CPU isolation (on an AP) between slices. First, per-flow bandwidth isolation can be performed on the agents using a token-bucket approach, but this only provides weak isolation on the physical layer, due to the dynamic characteristic of the wireless medium. Although modern WiFi cards are equipped with multiple queues to provide QoS, the assigned priorities and scheduling are hard to adjust. Hence, the FlowVisor approach of per-port queues does not suffice, and WiFi-specific QoS mechanisms need to be incorporated. Second, for agent CPU isolation, throttling control messages between the controller and agent does not suffice. This is because the performance of the pub-sub mechanism has a direct bearing on the effectiveness of a reactive application. If we throttle notifications being sent from an agent to the controller, it may negatively affect the decision-making at the application. We are currently exploring what the right design points are.

9 Conclusion

In this paper, we introduced Odin, an SDN framework for WiFi networks. Through the LVAP abstraction, Odin is well suited to address the complexities of the IEEE 802.11 protocol as demonstrated via the six common network services we have realized with it. Odin runs on top of *today's* commodity access point hardware without requiring client modifications, whilst being well-suited by design to take advantage of upcoming trends in physical layer virtualization and hardware extensions. Thus, with our publicly available prototype, we present one promising way to uniformly manage both wired and WiFi networks given the requirements of today's network operators. We are exploring this further by focusing on unified management of both wired and wireless resources.

Acknowledgments: We would like to thank our shepherd, Anthony D. Joseph, and the anonymous reviewers for their valuable feedback.

References

[1] Wi-Fi.org, http://shar.es/Qmnez, May 9th, 2014.

[2] Pica8, http://pica8.org/blogs/?p=201, May 9th, 2014.

[3] Deutsche Telekom. http://tinyurl.com/dtoffload.

[4] Meraki. https://meraki.cisco.com.

[5] Meru Networks. http://www.merunetworks.com.

[6] Swisscom Selects Aptilo. http://tinyurl.com/swisscomoffload.

[7] A. Mishra et al. A client-driven approach for channel management in wireless lans. In *IEEE INFOCOM 2006*.

[8] M. Bansal, J. Mehlman, S. Katti, and P. Levis. OpenRadio: a Programmable Wireless Dataplane. In *ACM HotSDN '12*.

[9] Y. Bejerano and S. Han. Cell breathing techniques for load balancing in wireless LANs. *IEEE ToMC*, June 2009.

[10] G. Bhanage, D. Vete, I. Seskar, and D. Raychaudhuri. Splitap: Leveraging wireless network virtualization for flexible sharing of wlans. IEEE GLOBECOM 2010.

[11] A. Cidon, K. Nagaraj, S. Katti, and P. Viswanath. Flashback: decoupled lightweight wireless control. In *ACM SIGCOMM '12*.

[12] Eduard Goma et al. Insomnia in the Access or How to Curb Access Network Related Energy Consumption. In *ACM SIGCOMM'11*.

[13] G. Bianchi et al. Maclets: active mac protocols over hard-coded devices. CoNEXT '12.

[14] Y. Grunenberger and F. Rousseau. Virtual access points for transparent mobility in wireless lans. In *IEEE WCNC 2010*.

[15] J. Herzen, R. Merz, and P. Thiran. Distributed spectrum assignment for home wlans. In *IEEE INFOCOM 2013*.

[16] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ToCS 2000*.

[17] L. Suresh et al. Demo: programming enterprise WLANs with Odin. In *Proc. SIGCOMM '12 (Demo)*.

[18] M.E. Berezin et al. Multichannel virtual access points for seamless handoffs in ieee 802.11 wireless networks. In *IEEE VTC Spring 2011*.

[19] R. Murty, J. Padhye, R. Chandra, A. Wolman, and B. Zill. Designing high performance enterprise Wi-Fi networks. In *Proc. NSDI '08*, 2008.

[20] R. Murty, J. Padhye, A. Wolman, and M. Welsh. Dyson: an architecture for extensible wireless LANs. In *Proc. USENIX ATC '10*.

[21] I. Papanikos and M. Logothetis. A study on dynamic load balance for IEEE 802.11b wireless LAN. In *COMCON '01*.

[22] R. Bolla et al. The potential impact of green technologies in next-generation wireline networks: Is there room for energy saving optimization? *IEEE Communications Magazine 2011*.

[23] S. Rayanchu, A. Patro, and S. Banerjee. Catching whales and minnows using wifinet: deconstructing non-wifi interference using wifi hardware. NSDI'12.

[24] R. Riggio, C. Sengul, L. Suresh, J. Schulz-Zander, and A. Feldmann. Thor: Energy Programmable WiFi Networks. In *IEEE INFOCOM '13 (Demo)*.

[25] E. Rozner, Y. Mehta, A. Akella, and L. Qiu. Traffic-aware channel assignment in enterprise wireless lans. In *ICNP 2007*, pages 133–143, Oct 2007.

[26] S. Shenker. An attempt to motivate and clarify sdn. http://www.youtube.com/watch?v=WVs7Pc99S7.

[27] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar. Can the production network be the testbed? In *OSDI '10*.

[28] J. Sommers, H. Kim, and P. Barford. Harpoon: a flow-level traffic generator for router and network tests. In *ACM SIGMETRICS '04/Performance '04*.

[29] Steven S. Hong et al. Picasso: Flexible RF and Spectrum Slicing. In *ACM SIGCOMM 2012*.

[30] L. Suresh, J. Schulz-Zander, R. Merz, A. Feldmann, and T. Vazao. Towards programmable enterprise WLANS with Odin. In *ACM HotSDN '12*.

[31] G. Tan and J. Guttag. Time-based fairness improves performance in multi-rate wlans. In *Proc. of USENIX'04*, Boston, MA, 2004.

[32] K. Tan, J. Fang, Y. Zhang, S. Chen, L. Shi, J. Zhang, and Y. Zhang. Fine-grained channel access in wireless LAN. In *ACM SIGCOMM 2010*.

[33] The Climate Group. SMART 2020: Enabling the low carbon economy in the information age. Technical report, 2008.

[34] V. Shrivastava et al. CENTAUR: realizing the full potential of centralized wlans through a hybrid data path. In *Proc. MobiCom '09*.

[35] V. Shrivastava et al. Pie in the sky: online passive interference estimation for enterprise wlans. NSDI'11.

[36] L. Yang, W. Hou, L. Cao, B. Y. Zhao, and H. Zheng. Supporting demanding wireless applications with frequency-agile radios. In *USENIX NSDI '10*.

[37] K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, and N. McKeown. The Stanford OpenRoads deployment. In *WiNTECH 09*.

[38] K. Yap, R. Sherwood, M. Kobayashi, T. Huang, M. Chan, N. Handigol, N. McKeown, and G. Parulkar. Blueprint for introducing innovation into wireless mobile networks. In *ACM VISA 2010*.

[39] P. Zerfos, G. Zhong, J. Cheng, H. Luo, S. Lu, and J. J. Li. DIRAC: a software-based wireless router system. In *MobiCom*, 2003.

HACK: Hierarchical ACKs for Efficient Wireless Medium Utilization

Lynne Salameh, Astrit Zhushi, Mark Handley, Kyle Jamieson, Brad Karp
University College London

Abstract

WiFi's physical layer has increased in speed from 802.11b's 11 Mbps to the Gbps rates of emerging 802.11ac. Despite these gains, WiFi's inefficient MAC layer limits achievable end-to-end throughput. The culprit is 802.11's mandatory idle period before each medium acquisition, which has come to dwarf the duration of a packet's transmission. This overhead is especially punishing for TCP traffic, whose every two data packets elicit a short TCP ACK. Even frame aggregation and block link-layer ACKs (introduced in 802.11n) leave significant medium acquisition overhead for TCP ACKs. In this paper, we propose TCP/HACK (Hierarchical ACKnowledgment), a system that applies cross-layer optimization to TCP traffic on WiFi networks by carrying TCP ACKs within WiFi's link-layer acknowledgments. By eliminating all medium acquisitions for TCP ACKs in unidirectional TCP flows, TCP/HACK significantly improves medium utilization, and thus significantly increases achievable capacity for TCP workloads. Our measurements of a real-time, line-speed implementation for 802.11a on the SoRa software-defined radio platform and simulations of 802.11n networks at scale demonstrate that TCP/HACK significantly improves TCP throughput on WiFi networks.

1 Introduction

In today's WiFi wireless networks, each time a sender wishes to transmit, it must first sense the medium to be idle for a randomly chosen interval. These random delays desynchronize would-be concurrent senders. To use a concrete example, Enhanced Distributed Channel Access (EDCA) in 802.11n [1] enforces an average idle period of 110.5 μs before a frame's transmission, whereas a 1500-byte payload itself lasts only 80 μs at 150 Mbps. Each frame's link-layer acknowledgment (LL ACK) consumes further channel capacity. As the physical-layer bit-rate increases but the pre-transmission idle period remains the same, this inefficiency only worsens. If a 600 Mbps 802.11n sender sent single frames in this fashion, it would only achieve 9% of the theoretical channel capacity. Moreover, WiFi senders back off exponentially after a failed transmission, and so incur even longer mean pre-transmission idle periods under contention, further reducing medium efficiency.

In an effort to amortize the significant overhead of medium acquisition over multiple data frames, 802.11n's MAC protocol batches multiple data frames into a single aggregate MAC protocol data unit (A-MPDU), and incurs only a single medium acquisition for each such batch. 802.11n further aggregates the LL ACKs for the data packets in a received A-MPDU into a single LL Block ACK. While batching helps one sender, TCP traffic is inherently *bidirectional:* a TCP receiver typically transmits a single TCP ACK packet for every pair of TCP data packets it receives. Not only do TCP ACKs incur further expensive medium acquisitions by the TCP receiver—they run the risk of colliding with the TCP data sender's transmissions as well.

WiFi's data frames elicit LL ACKs that the receiver sends without contending for the medium, as other would-be senders defer for an ACK frame's duration after hearing a data frame. We observe that this LL ACK is an ideal vessel for carrying TCP ACK information on the reverse path without incurring a costly medium acquisition. We name this overall cross-layer approach—in which a single transmission of feedback by a lower-layer protocol additionally carries feedback from a higher-layer protocol—Hierarchical ACKnowledgment (HACK). Though applying HACK to carry TCP ACKs in LL ACKs is conceptually quite simple, a robust design to do so must address several systems challenges. In this paper, we describe and evaluate such a design, TCP-over-HACK (TCP/HACK). Our contributions in this work include:

- We offer an analysis of the capacity of the 802.11n MAC protocol for TCP traffic as function of bit-rate, and the throughput gains theoretically achievable by avoiding medium acquisitions for TCP ACK packets.
- We describe TCP/HACK, a scheme that increases the WiFi MAC's efficiency by encapsulating TCP ACK information in WiFi LL ACKs. TCP/HACK fully supports 802.11n's batching of data packets and use of LL Block ACKs.
- We show how to efficiently encode the full range of TCP ACK information (*e.g.,* timestamp options, receiver window changes) within LL ACKs.
- We identify potential pathological interactions between TCP's reliability and congestion control mechanisms and WiFi's LL reliability protocol that would limit system throughput, and ensure that TCP/HACK avoids such interactions, without any changes to any node's

The research leading to these results has received funding under the EU's 7th Framework Programme, grant n° 257422, n° 287581, and European Research Council grant n° 279976. We gratefully acknowledge a hardware donation from the Microsoft Research Software Radio Academic Program.

TCP implementation.

- We offer an interface between the network device driver software and the network interface card (NIC) hardware that minimizes complexity in the NIC while allowing prompt sending of TCP ACK information in WiFi LL ACKs generated in response to WiFi data packets.
- Through an evaluation in simulation of up to 10 competing TCP flows on a 150 Mbps 802.11n network, we illustrate that TCP/HACK improves aggregate throughput up to 22% over TCP on "stock" 802.11n.
- Through an evaluation of a prototype online, wire-speed implementation of TCP/HACK for 802.11a on the SoRa software-defined radio platform, we illustrate that TCP/HACK improves aggregate throughput up to 32% over TCP on "stock" 802.11a.

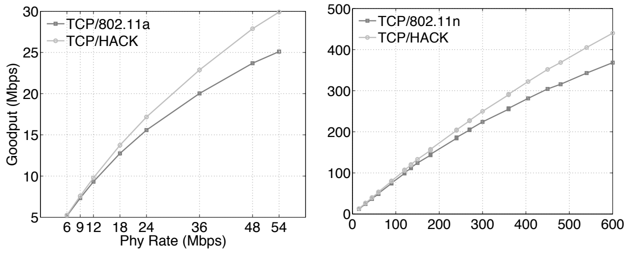
2 Problem and Design Goals

There are two distinct facets to improving the efficiency of the WiFi MAC layer for TCP transfers at fast bit-rates. First, we must understand the overhead of medium acquisition in WiFi 802.11a and 802.11n networks. How inefficient is the status quo, and what potential performance gains can one achieve by reducing the number of medium acquisitions? Second, we must articulate goals for our design to ensure that it meets the practical challenges of carrying feedback from a higher-layer protocol in a lower-layer one, as we propose to do in HACK. Such challenges arise because of the vagaries of wireless links (*e.g.,* frequent packet losses on links with poor signal-to-noise ratios), the potential for pathological interactions between TCP and the WiFi MAC protocol when optimizing across layers, and the constraints of real-world protocol stacks, network device drivers, and NICs. We now consider these two facets—medium acquisition overhead and practical design goals—in turn.

2.1 WiFi MAC Overhead

Consider a typical WiFi use scenario, where a single 802.11a or 802.11n client downloads a large file from a remote TCP sender. We assume throughout that the TCP receiver uses delayed ACK, and thus generates one TCP ACK packet for every two TCP data packets it receives.[1]

In Figures 1(a) and 1(b), the curves labeled "TCP 802.11{a,n}" show analytical predictions of the throughput a single TCP downloader achieves as a function of physical-layer bit-rate on lossless 802.11a and 802.11n networks, respectively. These analytical predictions are based on the parameters of the 802.11a and -n MACs. A detailed derivation of the capacity of the 802.11n MAC layer may be found in [6]; we do not repeat it here. The calculation for 802.11a is similar. (Figures 1(a) and 1(b) also show the improved throughput achieved by HACK, our modified 802.11 MAC protocol that carries TCP ACKs in link-layer ACKs, which we describe in Section 3.)



(a) (b)

Figure 1: Theoretical goodput for 802.11a (a) and 802.11n (b) rates. In (b), theoretical TCP/HACK achieves an 8% improvement on average over TCP/802.11n for physical rates lower than 100 Mbps.

Note that for "stock" 802.11a and -n, the achievable TCP throughput is a progressively smaller fraction of the physical layer bit-rate as the latter increases. Time spent on non-payload overhead for each medium acquisition is to blame. In 802.11a, these overheads include the durations of DIFS and the contention window (both before a data frame's transmission), the data frame's preamble, the SIFS interval between data frame and LL ACK, and the LL ACK itself.[2]

As noted earlier, 802.11n aggregates data frames into A-MPDUs so as to amortize medium acquisition overhead over many frames, and combines multiple LL ACKs into Block ACKs in response. The results in Figure 1(b) include the application of these techniques, and show that while they reduce 802.11a's overhead, TCP still suffers progressively greater throughput limitations *vs.* the physical-layer rate because of the overhead of medium acquisitions for TCP ACKs.

2.2 Design Goals

To work robustly in practice, TCP/HACK must meet several demands that arise from the constraints of a modern wireless host's networking software and hardware, some of which are particularly unforgiving.

Hard real-time deadlines A WiFi receiver must reply to a data packet with an LL ACK within SIFS, an interval defined in the 802.11a specification (for example) as 16 μs. That deadline is of course far too short to meet in host software, so WiFi NICs validate received frames and generate LL ACKs in hardware. TCP/HACK must comply with these same LL ACK deadlines imposed by today's

[1] Note that this assumption is the best case for the efficiency of the status quo WiFi MAC—were delayed ACK not used, a TCP receiver would generate twice as many ACK packets, and the WiFi MAC would incur significantly more medium acquisitions.

[2] 802.11n's parameter names and values differ slightly (*e.g.,* AIFS instead of DIFS); the overall scheme of per-medium acquisition overhead does not.

WiFi MAC. But if TCP/HACK is to enclose TCP ACK information in LL ACKs, the host TCP implementation cannot possibly generate a TCP ACK for a newly received TCP data packet within SIFS. To accommodate typical host protocol stack processing delays, TCP/HACK must allow the TCP ACK for a newly received TCP data packet to be enclosed within the LL ACK for a *different* TCP packet received later. Yet it mustn't unduly delay the return of an ACK to the TCP sender (see "cross-layer nuances" below).

Efficient encoding of general TCP ACK information The WiFi MAC reserves time on the wireless medium for a LL ACK to return after a data packet, so that other senders' transmissions do not collide with the LL ACK. It is important that TCP/HACK encode TCP ACKs in LL ACKs efficiently, to minimize the period of medium occupancy for these lengthened LL ACKs. The encoding for TCP ACKs must be compact yet allow the full generality of information that may potentially be found in a TCP ACK, (*e.g.,* TCP timestamp options, changes in receiver's advertised window, *&c.*) all of which is important to the correct and efficient operation of TCP.

Simplicity of NIC modifications TCP/HACK should not require any in-NIC intelligence about TCP packet headers or other TCP protocol details. Both at clients and APs, all TCP-aware processing must occur in the host software. We set this goal to minimize the complexity and thus the cost of the NIC, but also because we would like HACK to generalize to other higher layers than TCP such as SCTP [10] or DCCP [5]: if the NIC treats the feedback to be appended to an LL ACK as opaque bits that it needn't understand, then HACK should generalize in this way.

No changes to TCP TCP changes are difficult to standardize and difficult to deploy, as many widely used OSes ship with a single closed-source TCP implementation. Both at clients and APs, HACK-related functionality should be confined to the WiFi NIC's device driver (which is bound to the NIC's hardware design—*i.e.,* NIC hardware that supported HACK would routinely ship with a driver supporting HACK).

Avoid pathological cross-layer interactions Finally, it is important to note that TCP relies on a stream of TCP ACKs reaching the sender to maintain steady packet transmissions by the sender (and thus high throughput). TCP/HACK must not disrupt the timely return of correct TCP ACKs to the sender.

3 HACK DESIGN

We first offer an overview of TCP/HACK's design. We then explore nuances of the cross-layer interactions between TCP and 802.11n, which motivate refinements to TCP/HACK that improve robustness and performance. Finally, we consider the constraints of real-world systems software and NIC hardware, as well as of lossy wireless links, and flesh out the design of TCP/HACK into a fully practical system.

In the interest of brevity, we describe the design of TCP/HACK in the context of an 802.11 client acting as a TCP receiver while downloading via an 802.11 AP. Throughout, we refer to this downloader as the "client." Note, however, that TCP/HACK is a fully symmetric design—both the design and our implementation of it also work on TCP uploads by an 802.11 client.

3.1 HACK in Overview

Let us first consider how TCP/HACK works in the simpler case of 802.11a, without batching of packets into A-MPDUs. When a regular TCP client receives a TCP data packet, its network stack generates a TCP ACK and enqueues it for transmission by the WiFi NIC.

Under TCP/HACK, a client does not immediately enqueue a TCP ACK for transmission. Instead, the client compresses each TCP ACK and appends them to a compressed frame that it builds. When the next data packet from the AP arrives, the client encapsulates the compressed TCP ACK frame within the returning LL ACK, *effectively avoiding all medium acquisitions for the corresponding TCP ACKs.* The AP recognizes an "augmented" LL ACK, which it decompresses, reconstitutes the encoded TCP ACKs, and forwards them upstream.

Now let us consider 802.11n, where data packets can be aggregated into a single batched A-MPDU, and link-layer ACKs take the form of a Block ACK that includes a bitmap indicating which packets from the A-MPDU were received. On "stock" 802.11n during a TCP download the normal repeating pattern will then be:

1. one A-MPDU from AP to client containing TCP data
2. one Block ACK from client to AP
3. one A-MPDU from client to AP containing TCP ACKs
4. one Block ACK from AP to client

To eliminate medium acquisitions for TCP ACKs in 802.11n, we would like a TCP/HACK client to encapsulate all the TCP ACKs generated in step 3 in the Block ACK sent in step 2, and thus avoid step 4 entirely. In practice, the arrival of an A-MPDU containing a batch of TCP data packets will cause the client's OS to generate a burst of TCP ACK packets in step 3 *after* the Block ACK has departed for that A-MPDU. These TCP ACKs arrive at the client's transmit queue where they are compressed and concatenated, waiting for the arrival of the *next* A-MPDU from the AP. The client will then append this compressed frame to the Block ACK it sends the AP in step 2.

Although the description above is for downloads, the design is in fact symmetric; we envisage TCP/HACK as especially useful for wireless backup to LAN-attached

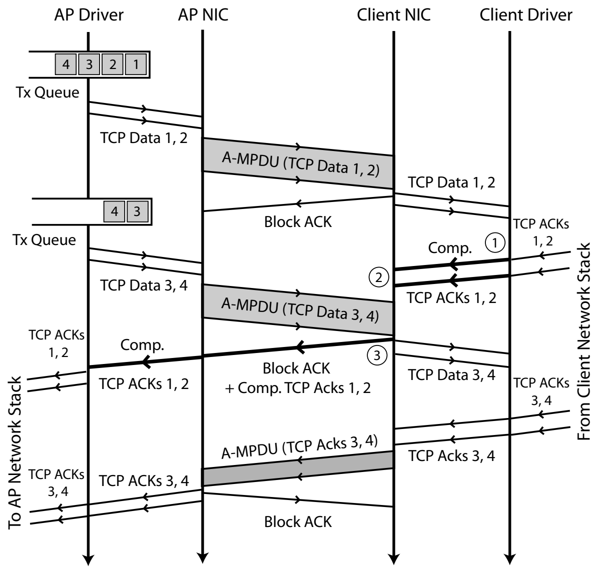


Figure 2: Interaction between A-MPDUs, Block ACKs and encapsulated HACK packets

storage, such as a Time Capsule.

3.2 Cross-Layer Nuances

We now refine our design to handle the subtle cross-layer interactions that arise between TCP and 802.11.

In principle, we would like to encapsulate TCP ACKs on the link-layer ACKs of the TCP packets they acknowledge. For example, if a batch containing TCP packets 1-64 arrives, the client would like to piggyback the TCP ACKs for packets 1-64 on the Block ACK for that batch. However, the 16μs SIFS interval between receiving data and sending the link-layer ACK or Block ACK is too short for the host's TCP stack to turn around the TCP ACKs, compress them, and DMA them to the NIC. For HACK to be practical, the compressed ACKs will have to wait until the next data arrives, and piggyback on its ACK or Block ACK. It turns out that this significantly complicates the dynamics of TCP/HACK and we will explore the consequences.

Figure 2 illustrates this process[3]. In response to a batch containing TCP packets 1 and 2, TCP ACKs 1 and 2 arrive at the client transmit queue too late to be carried on that batch's Block ACK. Instead, the TCP ACKs are compressed but not yet sent. When the next batch carrying TCP packets 3 and 4 arrives, its Block ACK can now carry the compressed frame with TCP ACKs 1 and 2. The AP then reconstitutes the full TCP ACKs and passes them up the network stack.

So long as TCP data packets continue to arrive, there is a steady stream of Block ACKs on which to piggyback compressed TCP ACK frames: all TCP ACKs are carried as HACK packets and no vanilla TCP ACK packets need to be sent. But what happens if no further data packets arrive? The client cannot retain the TCP ACKs for too long, or it will cause the TCP sender to time out and retransmit. Thus, after some time period, the client must send uncompressed vanilla TCP ACKs in the normal way. In Figure 2, TCP ACKs 3 and 4 meet this fate, and are in turn Block-ACKed.

Figure 1 summarizes the theoretical upper bound on TCP/HACK throughput on 802.11a (Figure 1(a)) and 802.11n (Figure 1(b)). The curves assume that the sender transmits the largest possible A-MPDUs,[4] that HACK manages to encapsulate all TCP ACKs in TCP Block ACKs, and that the compression is performed using the algorithm in Section 3.3. As the bit-rate increases, TCP/HACK significantly improves capacity, with a 20% improvement seen at 600 Mbps on 802.11n. In reality, the improvement can actually exceed that shown in the figure, as TCP/HACK can get closer to its bound than vanilla TCP can. This is due to collisions between TCP data packets and vanilla TCP ACK packets, a problem HACK sidesteps.

To HACK or not to HACK?

To maximize the benefits, TCP/HACK packets should be used whenever possible. But TCP ACKs must not be delayed when no more TCP data packets will arrive. How long should the client retain these TCP ACKs before giving up and sending them natively?

There are several reasons no more packets may arrive, including that the sender has stopped sending, but with 802.11n, the principal reason is the adverse effect of A-MPDUs on TCP's ACK clock. On a busy AP or during slow start, it is common for the entire TCP congestion window to be queued at the AP and then to be sent to the client in a single A-MPDU. An entire congestion window of TCP ACKs is generated and compressed, and these now sit at the client, waiting for the arrival of another incoming data packet so they can be send on its Block ACK. As the congestion window is full, this next packet never arrives and the connection stalls until TCP's retransmit timer fires. On 802.11a, which lacks aggregation, we don't often see this problem, but it is normal during slow start when 802.11n batching is used. We consider the following three different designs to address these concerns:

Explicit Timer A naive approach would be to have TCP/HACK time out and fall back to sending regular ACKs after a delay. In practice there is no good delay value that can be chosen, since the client cannot know the RTT and congestion window at the TCP sender, how the sender's packets will be spaced throughout the RTT, nor

[3] For simplicity it assumes that delayed TCP ACKs are disabled

[4] A-MPDU length is limited either by the 64 KByte A-MPDU bound or at lower bitrates by 802.11n's 4 ms transmit opportunity limit.

if the AP will suddenly start sending to another client.

Opportunistic HACK A more adaptive approach is not to explicitly delay TCP ACKs at all, but rather be opportunistic. When the wireless link is the bottleneck, the next downstream data batch will contend with the upstream TCP ACK batch. If the downstream batch wins, HACK can be used, but otherwise vanilla TCP ACKs will be sent. Such a design may often squander the opportunity to use HACK, but it has the virtue of seeming simple—until one considers the complexity of the NIC-network driver interface needed to implement it.

The MORE DATA Bit In Figure 2, initially there are four data packets queued at the AP. When the AP forms the first batch containing TCP data packets 1 and 2, it already knows more data will be sent to that client, as it already has packets 3 and 4 in its queue. So long as the AP has more packets queued than will fit in a batch, it knows that it is safe for the client to save up compressed ACKs waiting for the next batch. The AP simply tells the client that there is *more data* coming by setting the MORE DATA bit in the 802.11 header of the A-MPDU.[5] When the client sees this flag, it latches this state and will not transmit any more non-encapsulated TCP ACKs until the next data packet arrives, when it can use HACK to send them.

3.3 HACK in Practice

In the preceding section, we have presented a conceptual description of TCP/HACK, but several questions concerning the practicality of this conceptual design remain unanswered. First, how realizable is TCP/HACK given current systems and hardware? In particular, how should TCP/HACK's functionality be divided between a station's network interface card (NIC) hardware and NIC device driver? Finally, what manner of compression should TCP/HACK employ to reliably encode the TCP ACKs?

3.3.1 Driver and NIC Functionality

We realize TCP/HACK (including the MORE DATA mechanism) with very few changes to a station's 802.11 NIC. The main strategy is to implement the bulk of TCP/HACK within the NIC's driver, as we demonstrate using the example shown in Figure 2. Our discussion is in the context of a modern Linux wireless driver, such as the Atheros ath9k driver.[6]

AP (data transmission) The only modification needed to the AP when transmitting data packets is to set the MORE DATA flag when there are more packets remaining in the transmit queue for the same client.

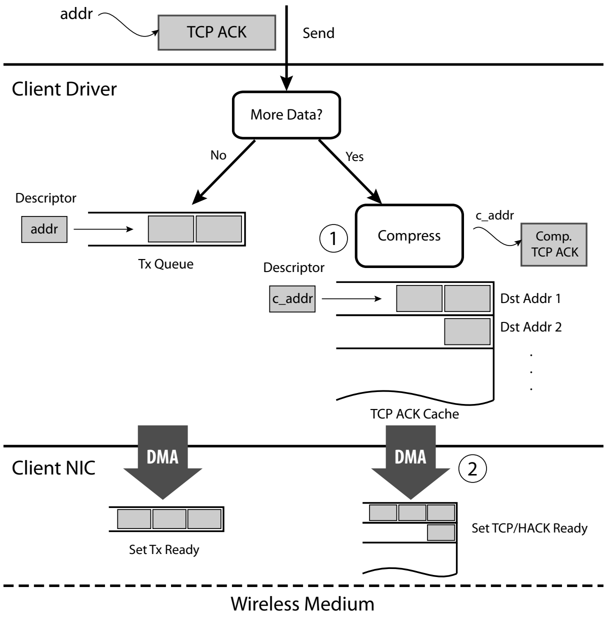


Figure 3: Client-side TCP/HACK compressing a TCP ACK, ready to be sent on the link-layer acknowledgment of the next frame.

Client The client's driver needs to determine when it can use TCP/HACK and when it must send TCP ACKs normally. In Figure 2, on receiving packets 1 and 2, the client's NIC also passes the MORE DATA state to the driver. The client TCP stack acknowledges the data, generating TCP ACKs 1 and 2, and puts them in the transmit queue at point ①.

Figure 3 shows what happens at points ① and ② from Figure 2 in more detail. If the driver is not in the MORE DATA state, it simply enqueues these ACKs normally. However, if MORE DATA is set, it compresses the arriving TCP ACKs and creates corresponding buffer descriptors. A separate buffer descriptor chain per destination address is needed to match compressed TCP ACKs with Block ACKs for that destination.

At point ② the driver DMAs the buffer descriptor chain to the NIC. The NIC maintains this table of compressed TCP ACK descriptors separately from normal transmission descriptors. Finally, the driver sets a flag in the NIC to indicate that TCP/HACK is ready.

Figure 4 shows what happens when the next batch from the AP arrives at the client. If the TCP/HACK flag indicates "ready," the NIC uses the corresponding descriptors to DMA the compressed TCP ACK frames to the card. It concatenates these frames, and appends them to the returning Block ACK at point ③. Recall that the NIC normally fires an interrupt when it receives data packets. In this case, the interrupt must also indicate whether the NIC succeeded in sending the compressed ACKs.

This design also copes with the race condition where

[5] This bit exists in stock 802.11 to assist with power saving. HACK uses this bit irrespective of whether power saving is enabled.

[6] http://wireless.kernel.org/en/users/Drivers/ath9k

the batch carrying packets 3 and 4 arrives with the MORE DATA flag not set before the driver has succeeded in conveying compressed TCP ACKs 1 and 2 to the NIC. In this case, the TCP/HACK "ready" check will fail. The NIC sends a normal Block ACK and signals to the driver a TCP/HACK failure in the receive interrupt. The driver now is free to re-enqueue the TCP ACKs on the transmit queue for normal transmission.

AP (ACK reception) Finally, the AP needs to recognize and decompress the "augmented" Block ACKs. The task of recognition falls to the AP's NIC, which extracts the compressed TCP ACK frame from the received Block ACK, adds it to the transmit complete report and interrupts to indicate transmit complete. The driver extracts the compressed TCP ACK frame, decompresses and reconstitutes the TCP ACKs, and forwards them upstream.

3.3.2 Compression

A critical component of the design is choosing a compression method for TCP ACKs. As 802.11a and -n transmit LL ACKs at one of the slower basic rates, *e.g.* 6 Mbps, it is desirable to minimize the size of the TCP ACK information appended to LL ACKs. Moreover, the 802.11a and -n MAC protocols' DIFS and AIFS intervals protect "stock" LL ACKs from collisions. Ideally, the compressed ACK information that HACK appends to LL ACKs should be short enough to fit within DIFS and AIFS, to avoid risking a collision.[7] We would like to leverage the redundancy within TCP and IP headers across consecutive TCP ACKs. Since most of the TCP/IP header fields remain static for a particular flow, they can be cached at the compression and decompression endpoints. To encode TCP and IP header fields reliably, TCP/HACK uses *Robust Header Compression (ROHC)* [8] to efficiently condense TCP/IP segments. ROHC supports the most popular TCP options like Timestamps and Selective Acknowledgments (SACK), and defines the notion of contexts, each with a particular identifier (CID). A context for TCP/HACK's purposes maps nicely to a particular TCP flow. In addition to caching static fields like the TCP/IP five-tuple at the endpoints, ROHC losslessly compresses the dynamic fields like the TCP Sequence and ACK numbers.

TCP/HACK-specific ROHC optimizations Since TCP/HACK applies ROHC in a specific context, we make the following simplifications:

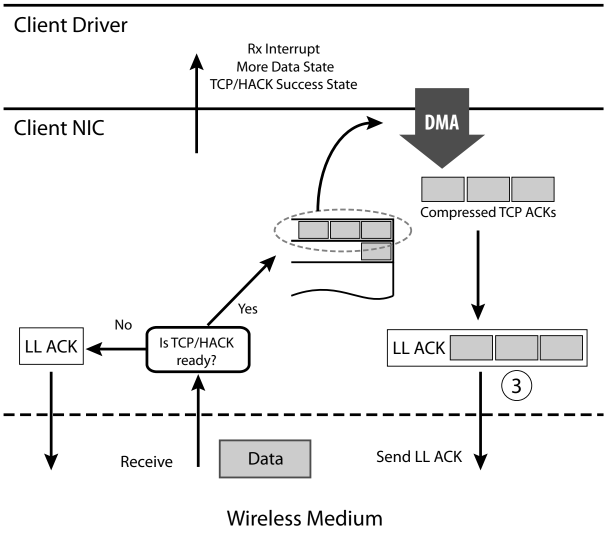


Figure 4: Client-side TCP/HACK receiving a batched frame from the air and including compressed TCP ACK frames in the corresponding link-layer acknowledgment.

1. We do not explicitly send Initialize-Refresh (IR) packets from the TCP client to the AP. To initialize a new context, the client can simply send uncompressed TCP ACKs outside of the TCP/HACK mechanism. The AP will consequently store the necessary state for the new context and assign it the correct CID.
2. The client and AP need not exchange any messages to agree upon a new CID for an emergent flow. Instead CIDs are computed independently at each endpoint. The client's driver on receiving a TCP ACK for a new flow computes the MD5 [9] hash over the ACK's 5-tuple and selects the lowest byte as the CID.
3. Compressed TCP ACK packets encapsulated within link-layer ACKs require a new mechanism to deal with losses outside of sending explicit ROHC feedback packets. We describe how TCP/HACK handles losses in Section 3.4.

With ROHC, a driver can shrink a TCP ACK to about 4 bytes, or even 3 bytes if the associated flow transmits a constant payload size (*e.g.* for large file downloads) [8].

3.4 Avoiding Cross-Layer Pathologies

The protocol we have described so far works well in a lossless environment. When applying HACK in low signal-to-noise ratio (SNR) regimes, decoding failures will cause packet drops. Any of the various packets sent by TCP/HACK may be dropped: TCP data packets, TCP ACKs, LL HACKs that contain LL Block ACKs and TCP ACK information, LL ACKs, *&c.* Under such losses, several concerns arise. To decompress headers correctly, ROHC requires that compression state at sender and receiver remain synchronized. Packet losses may cause loss of synchronization of this state, and in turn cause CRC failures on decompressed TCP ACK packets. Such loss of synchronization must not be persistent. We now describe

[7] In our simulations in Section 4.3, we find that 98.5% of the LL ACKs carrying ROHC-compressed TCP ACKs fit within AIFS for best-effort traffic. For the few that don't fit, the sender may either split the compressed TCP ACKs across multiple LL ACKs (ensuring each LL ACK is fully protected by AIFS) or it may send them all on a single LL ACK (risking a collision with a hidden terminal). Our simulator does the latter; there are no hidden terminals in the scenarios we simulate.

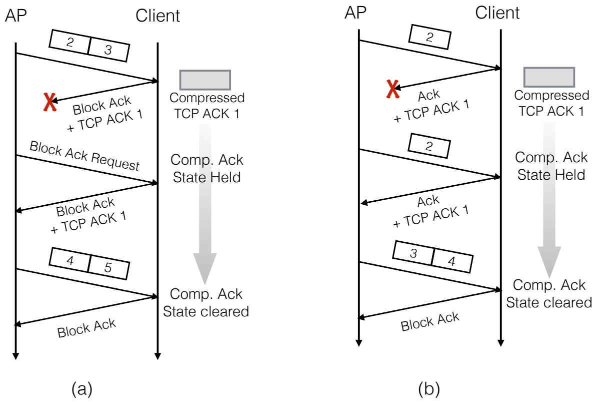


Figure 5: Coping with loss of (a) Block ACKs and (b) single LL ACKs by retaining TCP ACK state.

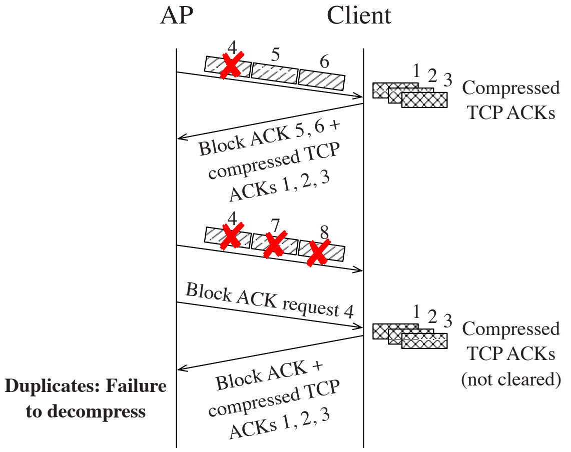


Figure 6: Retaining state: gap in sequence space.

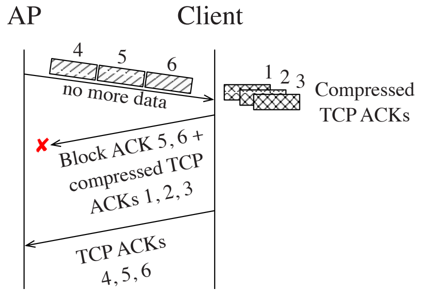


Figure 7: Flushing state: HACK-to-TCP ACK transition.

to restore lost synchronization quickly, to preserve the flow of TCP ACKs to the TCP sender.

Loss of LL ACK. First, consider the scenarios in Figures 5(a) and 5(b), where a Block ACK and single LL ACK carrying compressed TCP ACK information cannot be decoded, respectively. In both these scenarios, to deliver compressed TCP ACK(s) reliably, the client must retain them until it determines that its LL ACK (whether a Block ACK or a single LL ACK) has reached the AP. There is no such explicit indication from the AP, however. The client must enclose the same compressed TCP ACKs in all LL ACKs it sends to the AP until an *implicit* indication from the AP that the AP received the client's LL ACK. When the client has sent a Block ACK in response to an A-MPDU, as in Figure 5(a), receipt by the client of any subsequent A-MPDU (whether containing retransmitted MPDUs or not) indicates that the AP has received the client's Block ACK—if the AP has not done so, it must instead send a Block ACK Request. Alternatively, when the client has sent a single LL ACK in response to a single MPDU, as in Figure 5(b), the client can be certain that the AP has received its LL ACK upon receiving an MPDU with a greater MAC-layer sequence number—if the AP has not done so, it must instead retransmit the MPDU with the same MAC-layer sequence number. In both these cases, once the client has implicitly determined that its LL ACK has been received by the AP, it can safely discard any compressed TCP ACK information it has previously sent to the AP within that LL ACK.

Loss of retransmission. Since the ACKs themselves are not acknowledged, the ambiguity shown in Figure 6 can arise. The client cannot tell from the Block ACK request for 4 that the Block ACK for 5 and 6 was actually received. Thus it appends the compressed ACKs for 1,2 and 3 to the Block ACK response. Note that we cannot use the starting sequence number in the Block ACK Request as a signal that we have moved on to new data here because in this case it points to a gap in the sequence space, even though the rest of the aggregate is new.

The AP has already received and decompressed these ACKs, so its state is incorrect for decoding their retransmission. ROHC already has a mechanism to cope with duplicates—it has a master sequence number that increases monotonically. The lower 4 bits are normally included in each compressed packet. This is not sufficient for the first compressed TCP ACK packet carried in a Block ACK as an A-MPDU can carry 64 packets. We extend this first master sequence number to 8 bits, allowing the AP to discard duplicate compressed TCP ACKs and get back in sync.

Lost Block ACK, No More Data. Another corner case arises in Figure 7 when a Block ACK with compressed ACKs is lost, and the client needs to send vanilla TCP ACK packets because the last batch was not marked More Data. Here, the client clears any compressed TCP ACKs it has retained, and sends the next TCP ACK packet with a higher sequence number. TCP ACKs are cumulative and the upstream server will deal with the newer TCP ACK correctly even though there is a gap in received TCP ACK numbers.

Repeated loss of Block ACK. Finally, what happens when a Block ACK with compressed TCP ACKs is lost

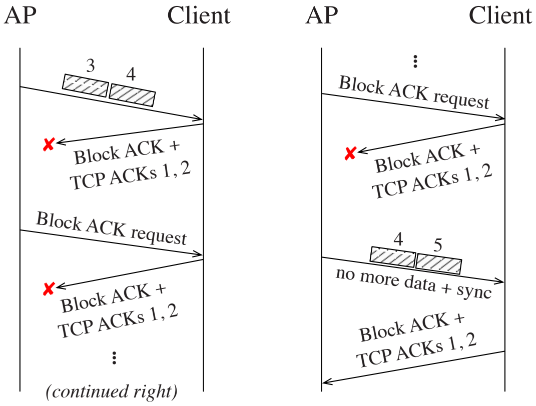


Figure 8: SYNC bit for retaining state.

repeatedly? Under normal 802.11n operation, the AP will continue to send Block ACK requests until it hits the retry limit, when it will give up and send the next batch of data. The client does not know that the AP has failed to receive the compressed TCP ACKs and, when it sees new data, it would normally discard the previously retained TCP ACK state. In this case, the AP explicitly notifies the client that it has moved on by setting a SYNC bit in the next batch's header. Upon seeing this bit set, the client doesn't discard the compressed TCP ACKs but rather appends them to the next Block ACK, as shown in Figure 8.

4 Evaluation

We evaluate TCP/HACK through a combination of simulation in ns-3 and experiments with a real-world implementation for the SoRa software-defined radio platform. We simulate TCP/HACK for 802.11n in ns-3, while our SoRa implementation is for 802.11a, as the public SoRa release does not support 802.11n.

4.1 SoRa Implementation

We implemented TCP/HACK including the More Data bit and ROHC compression for the SoRa user-level physical layer on Windows 7. Hardware limitations of our SoRa radio boards require us to run 802.11a in the 2.4 GHz band, but this does not affect protocol behavior.

One quirk of the SoRa platform bears mention. We have found that SoRa receivers sometimes return 802.11 link-layer ACKs later than the 802.11 specification's ACK timeout interval, causing spurious link-layer retransmits and backoffs. To avoid this performance hit, we increased the 802.11 ACK timeout to accommodate SoRa's late LL ACKs. The net effect of these delayed LL ACKs is that at 54 Mbps, our SoRa implementation only achieves 87% of the theoretical throughput across all protocols. We confirmed though simulation that this change does not significantly affect the relative benefit of TCP/HACK over regular 802.11a, but the absolute performance numbers are slightly lower.

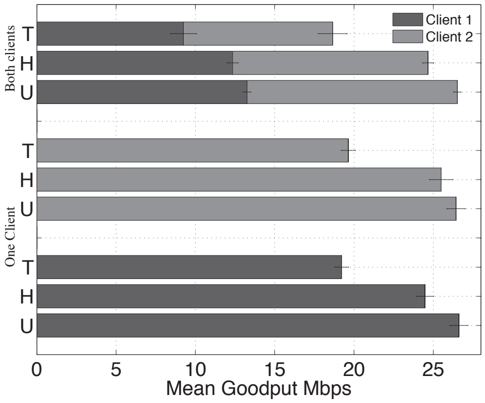


Figure 9: TCP throughput with stock 802.11a (T), TCP with HACK (H), and UDP (U) with stock 802.11a, with 1 and 2 clients.

Testbed Our three wireless nodes each have four-core Intel Core i7 CPUs, between 8–24 GB of RAM, and a PCI Express SoRa radio control board. One acts as the AP and the other two act as clients. We operate the SoRa interfaces in ad hoc mode to eliminate periodic beacon transmission. We run experiments on 802.11g channel 14 (2.484 GHz) in an open-plan office environment. We use *iperf* to generate TCP data streams with a 1500 byte MTU and send at 54 Mbps, the highest 802.11a rate.

4.2 SoRa Results

Besides demonstrating a successful implementation as evidence of TCP/HACK's practicality, we wish to answer several questions experimentally:

- Are TCP/HACK's capacity benefits in line with theoretical predictions?
- When an AP sends TCP flows to two clients, does TCP over 802.11a suffer collisions between clients' TCP ACKs, and if so, does TCP/HACK offer a performance benefit partly by eliminating such collisions?
- Do TCP/HACK's benefits come only from eliminating channel acquisitions and collisions, or are there other overheads that TCP/HACK eliminates?

Baseline Comparison Figure 9 compares the application-level throughput achieved by TCP/802.11a and TCP/HACK for bulk downloads, with UDP/802.11a for comparison. Each bar shows a different experiment: sending to one or both clients, using TCP over HACK, TCP over stock 802.11a or, as a control experiment, unidirectional UDP, which gives an upper bound on usable capacity. The data is the mean over five different 120-second runs; error bars show standard deviation.

Client 1's throughput is slightly less than Client 2's because it suffers a greater packet loss rate, even when only one flow is active. UDP's unidirectional data minimizes medium acquisitions, and achieves the greatest throughput possible on SoRa with link-layer ACKs enabled. In an ideal 802.11 MAC, UDP would achieve 30.2 Mbps; on SoRa, UDP averages 26.5 Mbps across the three experiments. SoRa's link-layer ACK delays alone reduce the attainable throughput to 28.1 Mbps, and our UDP measurements approach that figure.

If TCP/HACK encapsulated all TCP ACKs in LL ACKs, it would achieve almost the same throughput as UDP (though UDP's packet headers are smaller). In practice, TCP/HACK's single-client throughput of 25.0 Mbps (mean of C1 and C2) is very close to the UDP benchmark. TCP/802.11a only achieves 19.4 Mbps in this scenario. TCP/HACK improves performance by 29% and 32.2% in the one- and two-client cases respectively. Both TCP/HACK and TCP/802.11a are fair.

		UDP/ 802.11a	TCP/ HACK	TCP/ 802.11a
Client 1	no retries	99%	97%	87%
	1 or more	1%	3%	13%
Client 2	no retries	99%	98%	88%
	1 or more	1%	2%	12%
Both	no retries	99%	98%	86%
	1 or more	1%	2%	14%

Table 1: Percentage of frames successfully sent on the first attempt (no retries) and after one or more retries, when the AP is sending to Client 1 and Client 2 alone, and both clients at the same time, using UDP/802.11a, TCP/HACK, and TCP/802.11a.

Where do TCP/HACK's savings come from?

We note with interest that TCP/HACK improves throughput more than predicted analytically in Section 2.1. That prediction focused solely on saving medium acquisitions for TCP ACKs. In Table 1 we show the percentage of frames received after the first transmission, and the percentage that required one or more retransmissions. We see that TCP/802.11a experiences far more link-layer retransmissions than TCP/HACK or UDP/802.11a. These retransmissions occur because of collisions between TCP ACKs sent by clients and TCP data packets sent by the AP. TCP/HACK obviates most (but not all) of these TCP ACKs, and so significantly reduces the number of retransmissions needed. TCP/HACK not only eliminates costly channel acquisition overheads, but by encapsulating TCP ACKs in LL ACKs, also incurs fewer collisions.

	ACK count	ACK bytes	ACK_C count	ACK_C bytes	Comp. ratio
TCP/802.11a	9060	471120	0	0	(1)
TCP/HACK	10	520	9050	39478	12

Table 2: Conventional and compressed ACK counts, and compression rates of ROHC-compressed ACKs.

To understand other contributing factors in more detail, we ran an experiment where the AP transmits 25 Mbytes of data to a client using TCP/802.11 and TCP/HACK. By fixing the amount of work we can compare both protocols in time. The first two columns of Table 2 show the number of TCP ACKs sent as well as how many bytes were in those ACKs. The next two columns show the same figures for compressed ACKs, and the last column shows the compression rates ROHC acheives.

Reducing the number of transferred bytes is beneficial, but TCP ACKs are treated as regular data when sending over 802.11 wireless links and are sent at 54 Mbit/s in our experiments. LL ACKs, however, use the more robust 24 Mbit/s rate. To factor this in, we investigate how saved bytes translate into saved transmission time, together with TCP/HACK's impact on channel acquisition time and retransmission time.

	TCP ACK	ROHC	Acquire Channel	LL ACK overhead
TCP/802.11a	70 ms	0	1093 ms	456 ms
TCP/HACK	0.08 ms	13.1 ms	1.17 ms	0.46 ms

Table 3: TCP ACK time overhead breakdown for TCP/802.11 and TCP/HACK.

Table 3 shows time taken to send TCP ACKs (TCP ACK), time to send compressed TCP ACKs (ROHC), time spent waiting for channel before transmitting TCP ACKs (Channel) and extra time waiting for LL ACKs (LL ACK overhead). From the table, we see that most savings come from channel acquisition and LL ACK overhead.

Ideally LL ACKs are returned immediately after a SIFS time, but this is not always the case in the real 802.11 implementations. On SoRa we observe 37 μs on average of additional LL ACK overhead, while on two different commercially-available wireless NICs (the Atheros AR9300 and the Intel 5300) we measure 10.4-13.4 μs of LL ACK overhead, on average. While TCP/HACK benefits more from saving ACK overhead on SoRa than on the commercial cards, the benefit on commercial wireless hardware is still large. TCP/HACK not only eliminates channel overheads, it also reduces collisions and any additional LL ACK overheads incurred by the device.

SoRa and ns-3 Cross-Validation

To cross-validate our SoRa implementation against the ns-3 simulator, we simulated 802.11a in ns-3 with the same packet loss rate as that observed on SoRa (12% and 2% for TCP/802.11a and TCP/HACK, respectively). Since ns-3 returns LL ACKs immediately after SIFS, whereas SoRa incurs additional delay, ns-3 running TCP/802.11a achieves 22.4 Mbit/s *vs.* SoRa's 19.6 Mbit/s. After post-processing to eliminate SoRa's added LL ACK delay, we observe SoRa throughput of 22 Mbit/s, which matches simulation. Similarly, when simulating TCP/HACK in ns-3, we get 28 Mbit/s *vs.* SoRa's 25.5 Mbit/s. After accounting for SoRa's extra LL ACK delay, SoRa achieves 27.7 Mbit/s, which matches simulation.

4.3 Simulation Results

We now examine how TCP/HACK interacts with frame aggregation, with a larger number of clients than possible in our testbed. To this end, we implement A-MPDU support and TCP/HACK in ns-3. We evaluate both the opportunistic and MORE DATA variants of HACK described in Section 3.2 to verify that the the latter outperforms the former as hypothesized.

We simulate multiple WiFi clients scattered randomly within a circle of 10-meter radius centered on the AP. Our aim is to model the scenario where several clients connect via 802.11n WiFi to a server located nearby on a high-speed LAN. We present results modeling an 802.11n single-antenna setup using data packet and link-layer ACK bit-rates of 150 Mbps and 24 Mbps, respectively. The wired link between the server and the AP has a latency of one millisecond and a bit-rate of 500 Mbps.

To glean the benefits of the MORE DATA scheme, we would like AP's transmit queue to contain at least 126 packets per flow. We choose this number so that the AP may buffer of up to three batches of 42 packets per client, accounting for some variability in the A-MPDU size in the presence of TCP retransmissions. To avoid adverse "buffer bloat" effects [3], the transmit queue should not be too large in the case of one flow, but rather grow as the number of flows increases. A large buffer in our system would cause an excessive loss of packets when slow start overflows the buffer, with or without TCP/HACK. With ten clients, the AP's transmit queue would be 1260, which is reasonable since Linux drivers usually use buffer sizes of 1000 packets.

TCP/HACK *vs.* TCP/802.11n To determine the benefit of TCP/HACK and its constituent parts, we compute the aggregate goodput for TCP flows sending 1460 byte packets, averaged across five simulated runs per experiment. To mitigate phase effects with multiple clients, we stagger the starts of clients' downloads. As such, we compute the aggregate goodput over the steady-state portion of the runs, once all the clients have more or less exited slow start.

Figure 10 shows that UDP maintains a roughly constant goodput as the number of downloading clients varies, as expected. As a unidirectional protocol, UDP's performance is minimally affected by the number of clients competing for the link. In contrast, the goodput of TCP/802.11n decreases slightly as the number of downloading clients increases. Although the AP elicits TCP ACK packets from clients in turn, there is still a chance that two or more clients' TCP ACKs can collide, or that a TCP ACK can collide with a data packet from the AP.

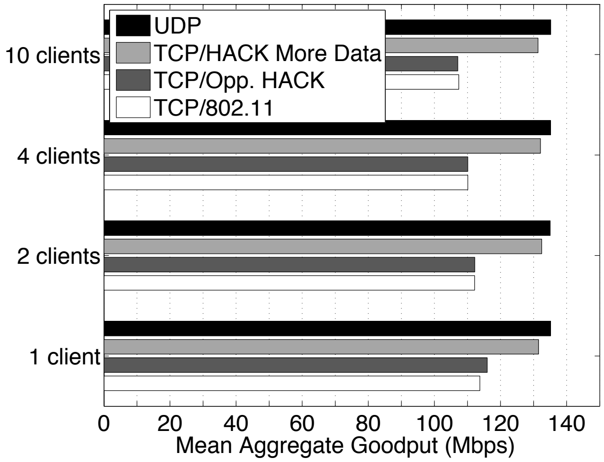


Figure 10: TCP goodput for different transmission schemes with 1–10 clients, and UDP for comparison.

These collisions account for the lower measured goodput than that predicted in Section 2.1.

We note with surprise that Opportunistic TCP/HACK does not significantly outperform TCP/802.11n: this most naïve implementation of HACK sends few compressed TCP ACKs in LL ACKs, and mostly regular TCP ACKs. It therefore does not achieve a TCP goodput closer to the physical rate.

Role of MORE DATA Bits We now turn our attention to the bars labeled "TCP/HACK More Data" in Figure 10. We observe that the MORE DATA variant of TCP/HACK achieves the most pronounced throughput gain over unmodified 802.11n. While simple, the MORE DATA mechanism is crucial to TCP/HACK's success in reducing medium acquisitions, and gives rise to goodput improvements between 15% for one client and 22% for ten clients at the physical rate of 150 Mbps.

Lossy Environment We next evaluate TCP/HACK under different SNR regimes. In addition to providing a wider spectrum of comparison between TCP/HACK and TCP/802.11n, these experiments will verify whether the HACK protocol with the properties described in Section 3.4 can indeed avoid any decompression CRC failures, or stalls due to recurring TCP timeouts.

We begin with a setup similar to that described above, and then place a single client at varying distances from the AP in order to simulate a decreasing set of SNRs. In lieu of simulating bit rate adaptation explicitly, at each particular distance we simulate a download of a 100 MB file at a rate selected from a range of 802.11n high throughput rates. This range corresponds to rates which are achievable using a 40 MHz channel, 400 ns guard interval and one antenna. The corresponding LL ACK rates are chosen from the set of basic rates (6, 12 and 24 Mbps) according to the rules outlined in the 802.11n specification. To

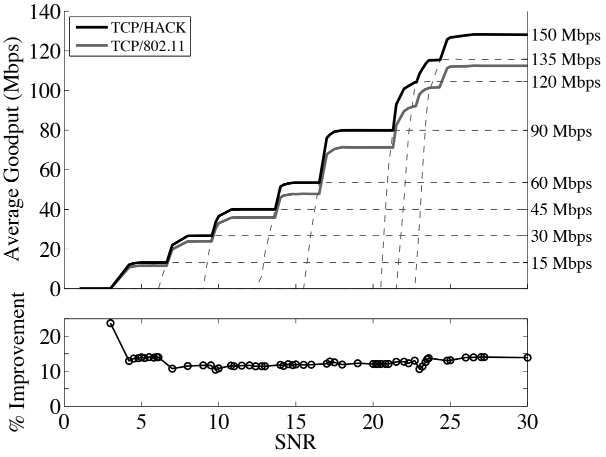


Figure 11: Envelope of average TCP goodput for TCP/HACK and TCP/802.11n under different SNR regimes and physical rates. The lower graph shows TCP/HACK's percent improvements over TCP/802.11n.

emulate a real system, we applied the 4 ms transmit opportunity limit to all transmissions, therefore limiting the size of A-MPDU packets for experiments using lower physical rates. At each distance/physical rate combination, we computed the average TCP goodput (including slow start) over five runs.

Figure 11 shows the average TCP goodput for TCP/HACK and TCP/802.11n. It plots a separate dashed curve per 802.11n physical rate for TCP/HACK. We use these curves to compute the envelope (in black), which indicates the best goodput achievable by an ideal bit rate adaptation algorithm. Similarly we plot the corresponding envelope for regular TCP/802.11n (the separate rate curves for TCP/802.11n are not shown).

Our simulations indicate that TCP/HACK functions correctly in a lossy environment and does not elicit any decompression CRC failures. Moreover, TCP/HACK improves TCP goodput by an average of 12.6% across the range of SNR values. Figure 11 shows that as the physical rate drops, the relative improvement increases slightly for the cases where the transmit opportunity limit reduces the number of packets a station can possibly transmit in an aggregate. Recall that 802.11n uses aggregation to amortize medium access costs, therefore we expect a better goodput gain for TCP/HACK over regular TCP at these rates. Similarly, as the physical rate increases past 90 Mbps, the overall improvement increases slightly to about 14%, because the 802.11n medium access delays now consume a larger portion of the transmission time relative to data.

Analytical Predictions *vs.* Simulations How well does the average TCP goodput measured in simulation match that computed analytically in Section 2.1? We extract the highest achievable goodput at each physical rate for both TCP/802.11 and TCP/HACK from the prior experiment, and plot these and the analytical predictions in Figure 12. As we expect, simulated goodputs are lower than the corresponding analytical predictions—the predictions do not model 802.11n collisions or retries, nor do they take into account TCP's retransmissions and congestion control.

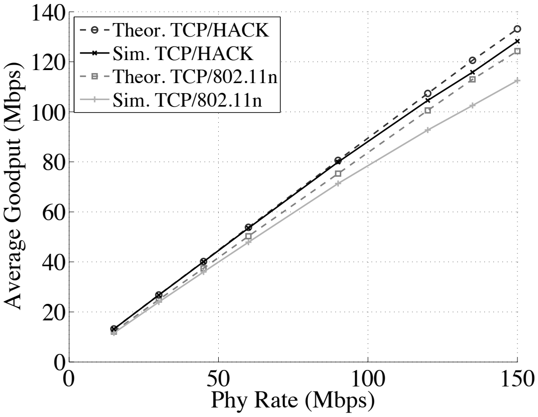


Figure 12: Theoretical and simulated TCP goodputs vs 802.11n physical rates.

Note, however, that the goodput improvement TCP/HACK offers over TCP/802.11n exceeds that predicted analytically. Since TCP/802.11n suffers more from collisions than TCP/HACK, its throughput suffers correspondingly more. TCP/HACK greatly reduces the collision rate by eliminating medium acquisitions for TCP ACK packets. At 150 Mbps, TCP/HACK offers a simulated goodput improvement of 14%, *vs.* the 7% improvement predicted analytically.

5 Discussion

Both batching using A-MPDUs and TCP/HACK help to reduce the time wasted on unnecessary WiFi medium acquistions. TCP/HACK relies on the More Data bit to know when it is safe to compress ACKs and wait for another packet on whose LL ACK to piggyback. A-MPDUs require sufficient packets in the AP's queue to gain efficiencies. With sufficient buffering at the AP and a large window, both work well. In such cases the wireless medium is busy, and efficiency is important. TCP/HACK can significantly reduce collisions when there are multiple senders by turning bidirectional TCP flows into unidirectional ones, reducing the number of contending hosts. However, if the traffic patterns are such that queues do not build in the AP or clients, there won't be enough packets to fill A-MPDUs or any remaining packets in the queue to allow the More Data bit to be set. Neither mechanism will work well in this case. Similarly, if an AP has very many clients, it may not buffer enough packets for each client for either mechanism to work well.

Longer batches improve utilization, but monopolize the medium for longer. 802.11e allows the AP to reduce medium acquisition latency by specifying a shorter maximum batch duration through the transmit opportunity limit. In such cases, we would expect TCP/HACK to help claw back some of the efficiency loss caused by limiting the maximum batch duration.

Sending a TCP timestamp option in the last TCP ACK of a batch would generalize the MORE DATA mechanism. The TCP sender would echo it, and the client could use receipt of the echo as an implicit ACK-of-ACK. When the client hasn't yet received a timestamp echo, it can reasonably expect further data to arrive, and thus delay sending TCP ACKs. We leave this for future work.

6 RELATED WORK

One approach to amortizing medium acquisition overhead across more data is narrow-band channelization. Since the effective data rate on each subband is much lower than that of wide-band 802.11, the time required for MAC-layer contention becomes smaller relative to the packet transmission time on a single subband, thus more effectively amortizing medium acquisition overhead across multiple packet transmissions. FICA [11] and WiFi-NC [2] take this approach. Both require redesigns of the physical and MAC layers. TCP/HACK is complementary: combining the two systems should yield greater medium efficiency than either system achieves alone.

WiFi-Nano [6] shortens the 802.11 contention slot time to 800 ns. TCP/HACK is again complementary: while WiFi-Nano reduces medium acquisition overhead, our proposal eliminates many medium acquisitions entirely.

Maranello [4] is a link-layer design for 802.11 wireless networks that incorporates sub-frame granularity checksums into link-layer acknowledgments, allowing the communicating pair to undertake partial packet recovery on corrupted frames. Unlike TCP/HACK, Han *et al.* implement Maranello partially on the firmware processor of a commodity 802.11 NIC, thus requiring access to the assembly source code of the firmware processor. While Maranello does not share the same networking goals as TCP/HACK, it does share systems context in terms of the hardware and software available to both designs. Like Maranello, TCP/HACK is realizable with very few changes to the NIC itself.

Of prior work in reducing channel acquisition overhead, Pang *et al.* [7] most closely resembles TCP/HACK, proposing that a client use a MAC-layer ACK to signal successful reception of TCP data. However, the designs they propose are only capable of communicating to the AP when a client observes a TCP ACK for the *same data packet just received.* The authors do not mention the possibility of the generation of a TCP ACK with a lower ACK number after a loss, and the link-layer feedback mechanism they propose is incapable of communicating any information to the AP other than "cumulative ACK for the data packet just sent to the client" or "no ACK for the data packet just sent to the client." As a result, these designs prevent the delivery of duplicate ACKs to the TCP sender, and prevent the use of fast retransmit, leaving only inefficient TCP timeouts. Furthermore, this work took place before the introduction of 802.11n, and as a result, does not consider the interaction with frame aggregation or block ACKs.

7 CONCLUSION

In this paper, we have described the design and implementation of TCP/HACK, a cross-layer acknowledgment design for TCP and the 802.11 MAC that eliminates most of the expensive medium acquisitions that TCP ACK packets require, significantly increasing TCP flows' wireless throughput. TCP/HACK improves throughput further when used with frame aggregation, yet offers significant throughput improvements without it. While frame aggregation and other previous approaches reduce the cost of individual medium acquisitions [11, 2, 6], TCP/HACK eschews many medium acquisitions entirely. It is thus complementary to these prior approaches. Our evaluations in simulation and in a real-world implementation confirm TCP/HACK's throughput improvements.

REFERENCES

[1] IEEE Standard 802.11-2012. Mar. 2012.

[2] K. Chintalapudi, B. Radunovic, V. Balan, M. Buettener, S. Yerramalli, V. Navda, and R. Ramjee. WiFi-NC: WiFi over narrow channels. In *NSDI*, Apr 2012.

[3] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the Internet. *CACM*, 55(1), 2012.

[4] B. Han et al. Maranello: Practical Partial Packet Recovery for 802.11. In *NSDI*, 2010.

[5] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *SIGCOMM*, 2006.

[6] E. Magistretti, K. Chintalapudi, B. Radunovic, and R. Ramjee. WiFi-Nano: Reclaiming WiFi efficiency through 800 ns slots. In *MobiCom*, 2011.

[7] Q. Pang, S. Liew, and V. Leung. Performance improvement of 802.11 wireless network with TCP ACK agent and auto-zoom backoff algorithm. In *Proc. IEEE VTC*, June 2005.

[8] G. Pelletier, K. Sandlund, L.-E. Jonsson, and M. West. *RObust Header Compression (ROHC): A Profile for TCP/IP*. RFC 6846, Jan 2013.

[9] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321, April 1992.

[10] R. Stewart. *Stream control transmission protocol*. RFC 4960, Sept. 2007.

[11] K. Tan, J. Fang, Y. Zhang, S. Chen, L. Shi, J. Zhang, and Y. Zhang. Fine-grained channel access in wireless LAN. In *SIGCOMM*, 2010.

Pythia: Diagnosing Performance Problems in Wide Area Providers

Partha Kanuparthy
Yahoo Labs

Constantine Dovrolis
Georgia Institute of Technology

Abstract

Performance problem diagnosis is a critical part of network operations in ISPs. Service providers typically deploy monitoring nodes at several vantage points in their network, to record end-to-end measurements of network performance. Network operators use these measurements offline; for example, to troubleshoot customer complaints. In this work, we leverage such monitoring infrastructure deployments in ISPs to build a system for near real time performance problem detection and root cause diagnosis. Our system works with wide area interdomain monitoring, unlike approaches that require data sources from network devices (SNMP, Netflow, router logs, table dumps, etc.). Operators can input operational and domain knowledge of performance problems to the system to add diagnosis functionality. We have deployed the system on existing monitoring infrastructure in the US, diagnosing over 300 inter-domain paths. We study the extent and nature of performance problems that manifest in edge and core networks on the Internet.

1 Introduction

End-to-end diagnosis of network performance is a significant part of network operations of Internet service providers. Timely diagnosis information is integral not only in the troubleshooting of performance problems, but also in maintaining SLAs (for example, SLAs of enterprise network and cloud provider customers). Knowledge of the *nature* of performance pathologies can also be used to provision network resources.

Performance diagnosis, however, is a challenging problem in wide area ISPs, where end-to-end (e2e) network paths traverse multiple autonomous systems with diverse link and network technologies and configurations. In such networks, operators may not have: *(i)* performance metrics from all devices comprising e2e paths, and *(ii)* labeled training data of performance problems. In this work, we explore a new system for troubleshooting based on domain knowledge, relying on e2e measurements and not requiring training data.

We leverage the monitoring infrastructure that ISPs deploy to record network health – consisting of several commodity hardware *monitors*. These monitors run low-overhead network measurement tools similar to *ping*, to record e2e delays, losses and reordering of monitored paths. Operators place monitors at vantage points in the network to maximize network coverage. An example of such deployments common in wide area academic and research networks is the perfSONAR software [6]; there are currently over 1,000 perfSONAR monitors spread across 25 countries [1]. Monitors running perfSONAR usually run the *One-Way Ping* (OWAMP) measurement tool, which probes over UDP every 100ms and records send/receive timestamps and sequence numbers.

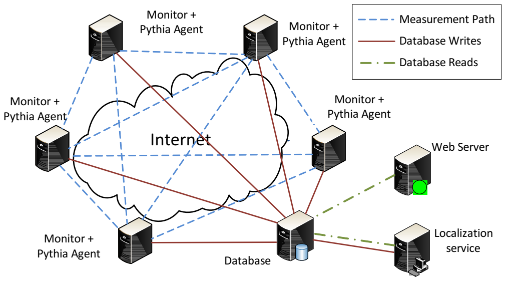


Figure 1: *A deployment of Pythia.*

Pythia works with e2e measurement streams recorded at monitors to do two tasks. First, it *detects* whether there is a performance problem on a monitored path at a given time. Second, it *diagnoses* the root cause(s) of detected performance problems. Pythia also *localizes* problems to network interfaces (using traceroutes recorded by monitors) [25]. In this paper, we focus on near-real time detection and diagnosis of short-lived and sporadic performance problems. Sporadic problems may not always be noticed by network operators, but can result in long-term failures and network downtime (e.g., gradual failure of an optical line card [1]).

A typical deployment of Pythia includes *lightweight* agent processes that run on monitors, and centralized database and web services (see Figure 1). Deployment can be done incrementally in ISP networks since deployment only involves adding the agent to new monitors. Pythia allows the operator to input diagnosis definitions using an expressive specification language as functions of *symptoms* in the measurements. This is useful since operators understand performance problems (and new ones that arise after network upgrades). Symptoms could include, for example, statistical changes in the delay timeseries, packet losses and/or packet reordering.

Pythia generates a *diagnosis forest* from the pathology specifications in order to reduce the number of symptoms that are tested on a measurement timeseries. The diagnosis forest reduces the agent's computation overhead at the monitor; and becomes important as the number of specifications grows with time or in large monitoring deployments. In practice, at a perfSONAR monitor probing a nominal set of 10 paths, each at 10Hz (default for

OWAMP), the Pythia agent receives a measurement every 10ms; we design a per-measurement agent run time of less than 100us. We describe common pathology specifications in Pythia that we wrote based on operator input: broadly related to congestion and buffering, loss nature, routing and reordering.

We make the following contributions in this paper. We design an efficient and scalable system and algorithms for real time detection and diagnosis (§2,4,5,6). We deploy Pythia in production wide area backbone networks to diagnose several pathologies. We use Pythia to do a first large-scale study of performance problems in edge and backbone networks (§9).

2 System and Monitoring Model

Pythia is a distributed system that works on top of existing monitoring infrastructure in ISPs. We design Pythia to scale to large monitoring deployments (potentially hundreds of monitors), without affecting the accuracy and timing of measurements taken at the monitors; and at the same time, with low network communication overhead. Pythia consists of a agent processes that run on monitors, a central database and a web server that renders real time diagnosis summaries (see Figure 1).

The agent is a lightweight process that performs two tasks. First, when the agent bootstraps, it parses the pathology specifications and generates efficient diagnosis code. Second, at runtime, agent reads measurements recorded at a monitor to detect and diagnose performance problems in near real time. It writes diagnosis output to the database. In order to minimize diagnosis-related traffic at agents, the agent runs diagnosis computation on the node that records the measurements; the agent computes diagnosis that requires information from other monitors using the database[1]. We design detection and diagnosis algorithms for the agent in Sections 4 and 5.

We consider a simple but general model of ISP monitoring. Suppose that N monitors are deployed by the ISP. Measurement tools in these monitors send active probes to other monitors, potentially over $N \times (N-1)$ end-to-end paths. For each monitored path $A \to B$, monitor A sends probes at a reasonably high frequency[2] to monitor B, and B records measurements of the probes; we do not require a specific probe sampling process. For each measurement probe that monitor B receives, a measurement tuple of sender and receiver timestamps, and a (sender) sequence number is recorded by B. The sequence number is incremented by one at the sender for each probe in the flow. We require loose clock synchronization (error margin of seconds) between the sender and receiver to correlate pathologies between monitors. Monitor M_i collects information about all probes sent to it; a lost packet is either "marked" by M_i as lost after a pre-defined timeout interval, or implicitly marked by a missing sequence number at M_i for that flow. We expect that a suitable *interface* exists on each monitor so that Pythia's agent can read measurements; the interface could be an API, a local cache, or simply files written to the disk.

Listing 1 Pathology specification grammar.

```
1. 'SYMPTOM' symptom
2. 'PATHOLOGY' pathology 'DEF' ( symptom |
   'NOT' symptom )
3. symptom → symptom_1 'AND' symptom_2
4. symptom → symptom_1 'OR' symptom_2
5. symptom → ( symptom )
6. 'PROCEDURE' symptom func
```

As an example, the perfSONAR monitoring software follows this model. It runs the OWAMP tool; an OWAMP endpoint A sends 40B UDP packets at 10Hz to endpoint B, and B records timestamps and sequence numbers in a file. Reading these, the Pythia agent at B computes one-way delay, loss and reordering for $A \to B$.

3 Pathology Specification

One of the design goals of Pythia is to allow the operator to *add* performance pathology definitions to the system based on, for example, domain knowledge of the network or the network performance. To do this, we would need a simple model for a pathology; the model would enable us to design a pathology specification language.

We model a performance pathology as *a unique observation on a set of symptoms*. Given a measurement timeseries $\mathscr{E}$, a symptom is a boolean-valued test $T(\mathscr{E}) : \mathscr{E} \to \{0,1\}$ such that T returns `true` if the symptom exists in the timeseries. A pathology is a logical expression on one or more symptoms. Pathologies differ either in the set of symptoms on which they are defined, or on the logical expression. Examples of symptoms include "interquartile of delays exceeds 100ms", "loss probability exceeds 0.5" and "non-zero reordering metric".

The pathology specification language is based on the pathology model. A pathology can be specified using the language as conjunction or disjunction operations on symptoms. The language also allows negations of symptoms. Listing 1 shows the pathology specification language grammar. An example of a pathology specification for a form of congestion is the following:

```
PATHOLOGY CongestionOverload DEF delay-exist
AND high-util AND NOT bursty-delays AND NOT
high-delayRange AND NOT large-triangle AND NOT
unipoint-peaks AND NOT delay-levelshift
```

The statement specifies a rule for the pathology `CongestionOverload` using seven symptoms.

[1] An alternative design is to ship measurements to a centralized compute cluster; this may not be feasible in ISPs due to network policies.

[2] The probing frequency for a path is expected to be high enough to observe short-lived performance problems, but at the same time, the average probing traffic is expected to be low-overhead.

The language keyword PROCEDURE specifies subroutine names for symptom tests. We define 11 pathologies in Pythia by default along with their symptom tests (§6). In order to add a new pathology definition, the user adds a pathology in the specification language and writes subroutines for boolean-valued tests that the pathology uses (or the user could reuse existing symptom tests). The parser replaces specifications of each pathology P containing disjunctions with multiple specifications of P that only contain conjunctions. Expressions of conjunctions allow us to represent specifications as a decision tree.

The diagnosis module generates a diagnosis forest, a compact intermediate representation of the specifications (§5), and generates diagnosis code from it.

4 Detecting Performance Problems

The problem of *detection* is the first step towards performance diagnosis. The agent process at a monitor reads path measurements of packet delays, loss and reordering to test *if there is a performance problem at a given point of time in a monitored path*. A challenge in detection is to define a generic notion of a performance problem – which is not based on the symptoms or pathologies. We define detection using a performance *baseline*.

We define detection as *testing for significant deviations from baseline end-to-end performance*. Suppose that we have a timeseries of measurements from a sender $\mathscr{S}$ to a receiver $\mathscr{R}$ (our methods are robust to measurement noise in timestamping). Under non-pathological (normal) conditions, three invariants hold true for a timeseries of end-to-end measurements of a path:

1. The end-to-end delays are *close to* (with some noise margin) the sum of propagation, transmission and processing delays along the path,
2. No (or few) packets are lost, and
3. No packets are reordered (as received at $\mathscr{R}$).

These invariants define baseline conditions, and a violation of one or more of these conditions is a deviation from the baseline. We implement three types of problem detection: *delay*, *loss* and *reordering* detection, depending on the baseline invariant that is violated.

The agent divides the measurement timeseries for a given path into non-overlapping back-to-back *windows* of 5s duration, and marks the windows as either "baseline", or as "problem" (i.e., violating one or more invariants). The agent then *merges* problem windows close to each other into a single problem window.

Delay detection: Delay detection looks for *significant deviations* from baseline end-to-end delays. We use the delay invariant condition (1) above, which can be viewed as a condition on modality of the distribution of delay measurements. Under normal conditions, the distribution of delay sample in the window will be unimodal with *most* of the density concentrated *around* the delay baseline $d_{\min}$ of the path (sum of propagation, transmission and processing delays). If there is a deviation from the delay baseline, the delay distribution will have additional modes: a *low* density mode around $d_{\min}$, and one or more modes higher than $d_{\min}$. The lowest mode in the delay sample's pdf is used as an estimate of the *baseline delay* for the window.

We use a nonparametric kernel smoothing density estimate [21] to find modes; with a Gaussian kernel (a continuous function) and the Silverman's rule of thumb for bandwidth[3] [21]. A continuous pdf enables us to locate modes (local maxima), the start and end points of a mode (local minima), and density inside a mode with a single pass on the pdf. The module also keeps track of the previous window's baseline for diagnosis of problems with duration longer than a window. To discard self-queueing effects in probing, the agent pre-processes the timeseries to check for probes sent less than $100\mu s$ apart.

We note that the delay detection algorithm has limitations; in particular, it may sometimes detect a long-term problem (minutes or longer) as multiple short-term problems. For example, a pathology such as a queue backlog (congestion) that persists for minutes may cause a level shift in delays – which could "shift" the baseline. The agent merges "problem" windows that are close to each other, and the operator may tune the window size to overcome this limitation. Our focus in this work, however, is on short-term performance problems.

Loss detection: Loss detection looks for *significant deviations from the baseline loss invariant condition* (2). Under normal conditions, the number of lost packets measured by monitors depends on several factors, including the cross traffic along the path, link capacities and the probing process. Since ISPs deploy low probing rates (e.g., 10Hz), Pythia looks at every lost probe. The loss detection algorithm marks a window as "problem" if the window contains at least one lost packet. Similar to delay detection, the agent merges "problem" windows close to each other into a single "problem" window.

Reordering detection: Reordering detection looks for *significant deviations from baseline reordering invariant condition* (3). The reordering module computes a *reordering metric R* for each 5s window of sequence numbers in received order, $\{n_1 \dots n_k\}$, based on the *RD* metric definition in RFC 5236 [9]. For each received packet i, it computes an *expected sequence number* $n_{\exp,i}$; $n_{\exp}$ is initialized to the lowest recorded sequence number in the timeseries, and is incremented with every received packet. If a sequence number i is lost, $n_{\exp}$ *skips* the value i (assume that the window starts with $i = 1$). A *reorder sum* is computed as: $\eta_{sum} = \sum_{i=1}^{k} \mid n_i - n_{\exp,i} \mid$. The reordering module estimates the number of reordered

[3] In case the bandwidth estimate is large, the delay distribution under the case of baseline deviation may be unimodal, but with a *large range* of delays under the mode.

packets in the window based on mismatch in the two sequence numbers: $\eta = \sum_{i=1}^{k} I\left[n_i \neq n_{\exp,i}\right]$. The reordering metric R for the window is defined as the ratio of the above (RFC 5236 [9]): $R = (\eta_{sum}/\eta)\, I\,[\eta \neq 0]$.

Note that R is zero if there was no reordering, and R increases with the *amount* of reordering on the path (i.e., both number of packets and how "far" they are reordered). Our goal is not to estimate the number of reordered packets (which may not have a unique solution), but to quantify the extent of reordering for a window of packets (and use R in diagnosis). The detection algorithm marks a window as "problem" if $R \neq 0$ for that window. The algorithm appends R to a reordering timeseries.

System sensitivity: Pythia provides a simple knob to the user to configure *sensitivity* of detection towards performance problems without asking the user for thresholds. This functionality reduces the number of problems that the system reports to the user, while ignoring relatively *insignificant* problems. Sensitivity is defined on a scale of one to 10, based on the fraction of delays higher than the baseline or the loss rate in the problem time window.

5 Diagnosis of Detected Problems

Diagnosis refers to the problem of finding the root cause(s) of a detected performance problem. The agent triggers diagnosis whenever it detects a (delay, loss or reordering) problem in the measurements of a path. The agent performs diagnosis by matching a problem timeseries with the performance pathology specifications, which network operators can extend using operational or domain knowledge. When the agent bootstraps, it generates diagnosis code from the specifications by building an efficient intermediate representation. We focus on the intermediate representation and code in this section.

A key aspect of diagnosis is to design algorithms that have low resource (CPU and memory) consumption, since the agent should not affect the measurement accuracy or probe timing at the monitor on which it runs. This becomes particularly important when tests for symptoms are resource intensive, or as the list of pathologies to test for gets large.

The diagnosis forest: A brute-force approach to diagnose a performance problem is to test the problem timeseries against *all* symptoms, and subsequently evaluate each pathology specification to find matching pathologies. This can be computationally expensive, since symptom tests could be expensive. Our goal is to *reduce the number of symptoms the agent tests for when diagnosing a problem*. An efficient way to do this is to build a decision tree from the pathology specifications. We evaluate the overhead of the algorithms in Section 8.

In order to generate diagnosis code, the agent generates an intermediate representation of the specifications: a *diagnosis forest*. A diagnosis forest is a forest of decision trees (or *diagnosis trees*). A diagnosis tree is an 3-ary tree with two types of nodes: the leaf nodes are pathologies, and rest of the nodes are symptoms. The tree edges are labeled either `true`, `false` or `unused`, depending on whether that symptom is required to be true or false, or if the symptom is not used. Hence, a path from the root node to a leaf node L in a diagnosis tree corresponds to a logical conjunction of symptoms for pathology L (specifically, symptoms that have non-`unused` edge labels).

Why not existing methods? There are several variants of decision tree construction methods such as the *C4.5* and *ID3*. These algorithms iteratively choose an attribute (symptom) based on the criteria of one that best classifies the instances (pathologies). They generate small trees by pruning and ignoring attributes that are not "significant". We found that existing construction methods are not suitable for the pathology specifications input for three reasons. First, pathologies may use only a small subset of symptoms (hence, we cannot treat `unused` as an attribute value in existing tree construction algorithms). In addition, not all outcomes of the symptom may be used in diagnosis; for example, Pythia does not include any pathologies which require the "loss exists" symptom to be `false`. Second, a pathology is required to be diagnosed using *all* symptoms in its specification (existing decision tree methods consider the *smallest* set of symptoms). Third, pathologies may exist *simultaneously* in an end-to-end path, and hence can be diagnosed in parallel (unlike a decision tree). Two pathologies can be diagnosed in parallel if both of their specifications match the problem timeseries.

Forest construction: The diagnosis forest is constructed in two steps. In the first step, the agent divides the set of pathologies $\mathscr{P}$ into disjoint subsets $\{P_1 \ldots P_k\}$, such that: (1) pathologies in each P_i use overlapping symptoms, and (2) sets P_i and P_j $(i \neq j)$ do not use common symptoms. Since no two members of the set $\{P_1 \ldots P_k\}$ use overlapping symptoms, we can run tests for P_i and P_j independently, and potentially have multiple diagnoses for a problem.

In the second step, the agent constructs a decision tree for diagnosing members in each pathology subset P_i. The initial tree is constructed such that the root node is the symptom that is *most frequently* used by pathologies, and such that the frequency of symptom usage drops as we go towards the leaves (Fig. 2). At the end of this step, the trees will contain *all* symptoms that are required to diagnose each pathology. We may, however, have some symptoms with `unused` outgoing edge labels.

Finally, the tree construction algorithm prunes as many unused symptoms as possible in the decision tree(s). This consists of two rounds of pruning on each tree (Fig. 2; the leaves are pathologies, and shades show symptoms). First, we ensure that each symptom node has

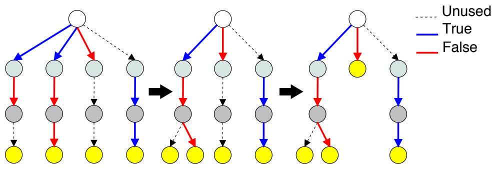


Figure 2: *Decision tree construction: pruning and merging.*

unique outgoing edge labels by *merging* edges with same labels. Second, we delete all edges with labels `unused` where feasible[4].

The default diagnosis rules configured in Pythia result in a forest of two trees – the first for diagnosis of delay and loss-based problems, and the second for diagnosis of reordering-based problems. The agent generates diagnosis code by traversing the decision forest. Each symptom node in a tree corresponds to a procedure call implemented by the agent for that symptom; and each leaf (pathology) node indicates a diagnosis output.

Unknowns: In practice none of the pathology definitions may match a detected problem timeseries. In such cases, the problem is tagged by the diagnosis logic as “unknown”. For each “unknown” problem, the agent checks whether the problem originated from a monitor, by looking at diagnoses across monitored paths in the system (see Section 6.6). In our data, we have observed that less than 10% of problems are tagged as “unknown”.

6 Common Diagnosis Specifications

We configure Pythia with a set of pathology specifications based on our domain knowledge of performance problems, and based on inputs from network operators. We expect that the network operator would add more based on her domain and operational knowledge of the network. Our goal behind the default choice of pathologies in Pythia is to provide the network operator with useful diagnosis information of the monitored networks.

In this section, we cover five classes of pathology specifications, and the statistical tests for matching the associated symptoms. We design boolean tests for symptoms by extracting salient and noise-resilient features of pathology models. The symptoms are defined over the measured timeseries for an end-to-end path - which includes delay, loss and reordering measurements. Table 1 lists the symptoms we test for. Some of the symptoms use domain knowledge-based thresholds, and can be fine-tuned by the operator.

Our network path model is as follows. An end-to-end path consists of a sequence of store-and-forward hops having a limited buffer size. We do not assume that links are work conserving, FIFO, or of a constant capacity (for example, 802.11 wireless links violate these assumptions). Monitoring hosts may induce delays, losses and “noise” in measurements.

6.1 End-host Pathologies

Our experience deploying Pythia on production monitors showed occurrence of short-term end-host pathologies – significant delays induced by effects inherent to commodity operating systems and userspace tools. Such pathologies occur at the monitoring nodes on which a Pythia agent runs. End-host pathologies may also refer to significant delays induced due to measurement application behavior (e.g., due to delays in timestamping or delays in sending probe packets). End-host effects may not be useful to the network operator; however, it is important for Pythia to identify and discard them, and not report them as pathological network delays[5].

End-host effects: A common artifact of commodity OSes is *context switches*. A busy OS environment may lead to significant packet *wait* delays at the sender and/or the receiver-side measurement end points. For example, these could be delays: *(i)* after a userspace `send()` call till packet transmission, or *(ii)* after the network delivers a packet to the OS until userspace `recv()` call (and corresponding timestamping).[6]

We can model an end-host induced delay symptom as follows. We model the buffered path from a measurement application to the NIC buffer (and the reverse path) as a single abstract buffer. Under normal conditions, this buffer services packets as they arrive from the measurement tool (or from the network). Under pathological conditions (e.g., when the OS is busy scheduling other processes or not processing I/O), the buffer is a non-work conserving server with “vacation periods”. If a vacation period of W is induced while packet i is being served, successive packet arrivals will see steadily decreasing wait delays (T_i is the send timestamp of i):

$$d_{i+k} = \max\left\{W - [T_{i+k} - T_i], 0\right\} \tag{1}$$

at the end-host (assuming the other packets do not see new vacation periods, and no other sources of delay variation). This behavior manifests in end-to-end delay measurement timeseries as a “triangular peak” symptom of height W, and the duration of this peak is also W.

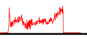
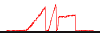
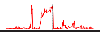
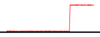
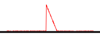
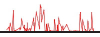
It can be argued that a vacation period could be a burst of cross traffic that arrived in the inter-probe duration δ. We choose our threshold for W to avoid matching such cases. Suppose that a burst arrived at a rate λ at a link of

[4] More specifically, for each edge $A \rightarrow B$, we delete A and move B upwards if (1) $A \rightarrow B$ has label `unused` (i.e., symptom A is unused in diagnosis of the sub-tree of A), and (2) B does not have any siblings.

[5] An alternative approach is to tackle end-host pathologies by rewriting the monitoring tool to reduce OS-level noise; e.g., by running in kernel space. It is, however, not feasible to do this in production.

[6] Note that context switches may also occur in network devices due to wait periods when the OS resources are busy; in practice, the likelihood is much higher in end-hosts, since they may be running multiple resource intensive processes (other than measurement tools).

Table 1: *Default symptoms and their boolean-valued tests. Tests take input delay sample $\mathscr{D} = \{d_1 \ldots d_n\}$, and the estimated baseline delay is b (both in ms). Tests work on a reordering metric timeseries $\mathscr{R} = \{R_1 \ldots R_l\}$. $I(x)$ is the 0-1 indicator function, and $m(\ldots)$ is the sample median. The default thresholds are tuned using empirical observations on perfSONAR data.*

Symptom	Sample	Boolean-valued Test
High delay utilization		More than 50% are *large* delays: $\sum_{i=1}^{n} I(d_i > b+1) > 0.5n$
Bursty delays		Largest delay hill duration less than 70% of delay hill duration
Extreme delay range		Very small: $\mathscr{D}_{0.95} - \mathscr{D}_{0.05} < 1$ms; very large: $\mathscr{D}_{0.95} - \mathscr{D}_{0.05} > 500$ms
Delay-loss correlation		For a lost packet i: $d_j > b+1$ms, for majority of $j \in [i-5, i+5] - \{i\}$
Small loss duration		All packets between i and $j > i+1$ are lost, and $T_j - T_i > 1$s
Delay level shift (LS)		Estimate LS: first/last $d_i < b \pm 10$ms; 10% points before/after LS
Large triangle		Sudden rises in delay over 300ms: $\sum_{i=1}^{n} I(d_{i+1} - d_i > 300) < 0.1n$
Single-point peaks		Median of neighbors of i:$d_i > b+1$ms ≈ median of d_j:$d_j \le b+1$
Reordering shift		One-proportion test for: $\sum_{i=l/2+1}^{k} I\left(R_i > m(R_1 \ldots R_{l/2})\right) = 0.5(l/2)$

capacity C. If the delay increase was *due to cross traffic*, we have the following condition for the queue backlog: $\left(\frac{\lambda - C}{C}\right)\delta \geq W$; in other words: $\lambda \geq \left(1 + \frac{W}{\delta}\right)C$. In our monitoring infrastructure, $\delta = 100$ms; so we can define a threshold for W by choosing an upper bound for the input-output ratio λ/C. We use a ratio of 4, giving us $W \geq 300$ms in case of an end-host pathology.

Depending on the magnitude of the vacation period W, we can have two end-host pathology symptoms. First, if W is of the order of 100s of milliseconds (e.g., when the receiver application does not process a probe in time), we will observe a "large triangle" delay signature (see Table 1), described by Equation 1[7]. Second, if W is much smaller – of the order of 10s of milliseconds (typical duration of context switches in a busy commodity OS) – and if the inter-probe gap is close to W, the delay symptom is a set of "single-point peaks": delay *spikes* that are made of a single (or few) point(s) higher than the baseline delay. The number of spikes is a function of the OS resource utilization during measurement.

6.2 Congestion and Buffers

Network congestion: We define congestion as a *significant* cross traffic backlog in one or more queues in the network for an extended period of time (few seconds or longer). Pythia identifies two forms of congestion based on the backlogged link's queueing dynamics. First, congestion *overload* is a case of a *significant and persistent* queue backlog in one or more links along the path. Overload may be due to a single traffic source or an aggregate of sources with a persistent arrival rate larger than the serving link's capacity. The congestion overload specification requires a high *delay utilization* above the baseline, and a traffic aggregate that is *not bursty*.

Second, *bursty* congestion is a case of a significant backlog in one or more network queues, but where the traffic aggregate is bursty (i.e., high variability in the backlog). We use the term "bursty" to refer to a specific backlog condition that is *not persistent*. The bursty congestion specification requires a high *delay utilization*, and "bursty" delays, i.e., the timeseries shows multiple delay *hills* each of reasonable duration. Both congestion specifications require that the timeseries does not show end-host pathology symptoms.

Buffering: A buffer misconfiguration is either an over-provisioned buffer or an under-provisioned buffer. Over-buffering has the potential to induce large delays for other traffic flows, while under-buffering may induce losses (and thus degrade TCP throughput). Pythia diagnoses a path as having a buffer that is either over- or under-provisioned based on two symptoms. First, the delay *range* during the problem is either too large or too small. Second, an under-provisioned buffer diagnosis requires that there is *delay-correlated* packet loss during the problem (see Section 6.3). We do not make any assumption about the buffer management on the path (RED, DropTail, etc.). We choose thresholds for large and small delay ranges as values that fall outside the typical queueing delays on the Internet (the operator can tune the values based on knowledge of network paths).

6.3 Loss Events

Random / delay-correlated loss: Packet losses can severely impact TCP and application performance. It is useful for the operator to know if the losses that flows see on a path are *correlated* with delays - in other words, delay samples in the neighborhood of the loss are larger than baseline delay. Examples of delay-correlated losses include losses caused by buffering in a network device, such as a full DropTail buffer, or a RED buffer over the

[7] We assume that the inter-probe gap is much lower than such W.

minimum backlog threshold.

On the other hand, *random losses* are probe losses which do not show an increase in neighborhood delays. Random losses may be indicative of a potential physical layer problem, such as line card failure, bad fiber or connector, or a duplex mismatch (we defined these based on operator feedback [2]). In theory, a random loss is the conditional event: P[loss | delay increase in neighborhood] = P[loss], where the *neighborhood* is a short window of delay samples. In practice, we may not have sufficient losses during the problem to estimate the probabilities; hence we look at the delay neighborhood[8]. Our loss-based pathologies may not have one-to-one correspondence with a root cause; however, operators have found them useful in troubleshooting [2].

Short outage: Network operators are well aware of long outages. They may overlook infrequent *short* outages, possibly caused by an impending hardware failure. A short outage can disrupt existing communications. Pythia diagnoses loss bursts that have a small loss duration (one to few seconds) as short outages.

6.4 Route Changes

The symptom of a route change is a *level shift* in the delay timeseries (and possibly, packet loss during the level shift). Routing events could be either long-term route changes, or route flaps. Pythia currently diagnoses long-term route changes. It does so by finding significant changes in the baseline and in the propagation delay. Note that delay level shifts can also occur due to clock synchronization at the monitors; Pythia currently reports delay level shifts as "either route change or clock synchronization". We do not support identification of clock synchronization events[9].

6.5 Reordering Problems

Reordering may occur either due to a network configuration such as per-packet multi-path routing or a routing change; or it could be internal to a network device (e.g., switching fabric design in high-speed devices) [4]. Reordering may not be a pathology, but it can significantly degrade TCP performance. If it exists, reordering will be either *persistent and stationary* (e.g., due to a switching fabric or multi-path routing), or *non-stationary* (e.g., routing changes).

Pythia diagnoses the above two types of reordering. The detection logic computes a reordering metric R for each time window of measurements (Section 4). R is zero if there is no reordering, and it increases with the *amount* of reordering on the path. Pythia diagnoses reordering non-stationarity by looking at the set of 10 most recent reordering measurements. Pythia uses the "reordering shift test" to diagnose non-stationarity (or stationarity) in reordering (see Table 1).

6.6 Unknowns

In practice, there may be detected problems that do not match any of the input pathology specifications. We call such problems as "Unknown" problems. When an agent finds that a problem is unknown, it performs an additional check across the monitored network to diagnose if the problem is a result of an end-host (monitor) pathology. Specifically, the agent checks whether *a significant number of paths ending at/starting from its monitor show a performance problem at the same time*. It does this by querying the Pythia database. The agent tags all unknown problems as end-host pathologies if a majority of paths were diagnosed as having an "Unknown" or an end-host problem.

7 Live Deployment

Many wide area ISPs consist of geographically distributed networks connected using inter-domain paths. Monitoring infrastructure in such ISPs consists of nodes in the constituent networks. For example, the US Department of Energy's Energy Sciences Network (ESnet), a wide area research backbone, connects several stub networks, each hosting several perfSONAR monitors.

We deploy Pythia on a wide area perfSONAR monitoring infrastructure that spans seven ESnet monitors across the US, a Georgia Tech monitor and monitors in 15 K-12 school district networks in GA, USA[10]. The current deployment uses the default corpus of 11 performance pathology definitions, some of which were formulated based on ESnet and Internet2 operational experience. We are in the process of expanding Pythia to several monitors in ESnet, Internet2 and other networks.

Our live deployment showed some interesting patterns from K-12 networks. We found using Pythia that in a typical week, about 70% of network-related pathologies are related to congestion, leading to packet losses. In particular, about 29% of the problems are due to traffic burstiness. Pythia also found that about 5% of the problems are packet losses not related to buffering, which may be due to physical layer-related problems. Moreover, Pythia's localization shows that almost 80% of the network interfaces are affected by one or more performance problems. Pythia also showed diurnal and weekday-related congestion patterns. Pythia's findings confirm with a prior study on the same K-12 networks in 2010 [18]. We use monitoring data to do a large-scale study of pathologies in Section 9.

[8] For a loss burst, Pythia considers delays before and after the burst.

[9] Identification of clock sync. events is an expensive operation. It can be done, for example, at a monitor M by correlating delays from all timeseries destined to or starting at M; if there is a clock sync, *all* timeseries will show a level shift at about the same time.

[10] The infrastructure is a part of the Georgia Measurement and Monitoring (GAMMON) project, which aims to assess the feasibility of online learning requirements of Georgia Department of Education.

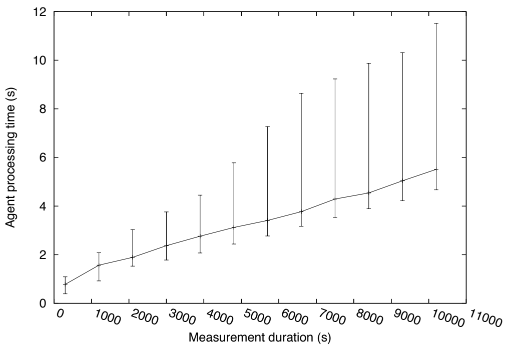


Figure 3: *Agent run time vs. input measurement duration: median, 5th and 95th percentile across 100 sample measurement files, excluding database commit times.*

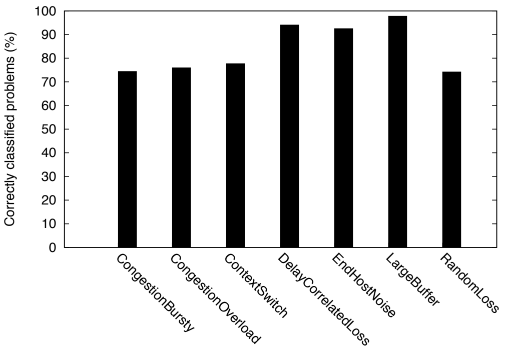


Figure 4: *Validation of diagnosis logic: classification accuracy of different diagnoses. Datasets include academic, research, commercial and residential broadband networks.*

8 Monitoring data and Validation

In the remainder of the paper, we look at how the different performance pathologies that Pythia diagnoses (§6) manifest in core and edge networks in the Internet. In order to do this, we collect data from production monitoring deployments in two backbone networks and leverage a popular measurement tool run by home users; we run the agent on the data. We collect data since we use it for offline validation and accuracy estimation.

Datasets: We collect data from production wide area monitoring deployments in backbone networks, and we build our own monitoring infrastructure to collect data from edge networks. We use four sources of data:

Backbone: This is data from production perfSONAR monitoring infrastructure in ESnet (33 monitors, 12 days) and Internet2 (9 monitors, 22 days). This data represents high-capacity, wide area inter-domain networks.

Residential: We use residential user-generated data from ShaperProbe [11]. ShaperProbe includes a 10s probing session in upstream and downstream directions. We consider 58,000 user runs across seven months in 2012.

PlanetLab: We build and deploy a full-mesh monitoring infrastructure similar to perfSONAR, using 70 nodes. We collect data for 12 hours in March 2011. We monitor commercial and academic networks.

Our data comes from 40 byte UDP probes, with an average sampling rate of 10Hz per path. OWAMP (perfSONAR) uses a Poisson process, while ShaperProbe and PlanetLab tools maintain a constant packet rate. All monitored networks have wide area inter-domain paths.

Agent overhead: We first measure the run time of detection and diagnosis algorithms in the agent. Figure 3 shows the agent run time as a function of input size – duration of measurements for a path – on a 3GHz Xeon processor. We use 100 randomly picked measurement timeseries from ESnet and Internet2. On an average, the agent takes about $60\mu s$ to process a single e2e measurement. Hence, the agent can handle large monitoring deployments (i.e., several measurement paths per monitor).

Validation: It is hard to validate an inter-domain diagnosis method, since it is not feasible to get systematic "ground truth" diagnoses (labeled data) of performance problems across the network. This is further complicated by our focus on *(i)* short-term problems which are typically unnoticed by network operators, and *(ii)* wide area ISPs, which include problems from multiple networks and inter-domain links.

In order to overcome paucity of labeled data, we use manually classified pathology data. We first run Pythia to find and classify performance problems in the data into different pathology types. For each of the four data sources, we choose a uniform random sample of 10 problems from each pathology type. We select a total of 382 problems for manual classification. Note that we would not be able to evaluate the false negative detection rate; it is infeasible to find problems that go undetected given the size of our data. Our detection methods, however, are not based on symptoms, and hence do not introduce systematic biases towards/against certain pathologies.

We manually observe the delay and loss timeseries of each problem and classify it into one or more pathologies (or as "Unknown"). For each problem, we mark as many valid (matching) pathologies as we see fit. We consider this as our ground truth. This approach has limitations. First, a high delay range in the timeseries (e.g., due to few outliers) may visually *mask* problems that occur over a smaller delays. Second, if the problem is long-duration, it may be hard to visualize small timescale behavior (e.g., a small burst of packet losses will be visualized as a single loss). Third, if there are unsupported diagnoses (e.g., short-term delay level shifts), the matching pathology definitions would be wrong, while the ground truth would be correct. Finally, the ground truth generation does not include "cross-path" checks that Pythia uses. An example of such checks is for diagnosing end-host pathologies in "Unknown" problems (§6.6).

To validate, we compare the diagnoses generated by Pythia for each problem with the manually classified di-

Table 2: *Occurrence of non-network problems in different datasets. Acronyms: "E.H.N." - "EndHostNoise", "C.S." - "ContextSwitch", "Unk." - "Unknown".*

Dataset	# Problems	E.H.N.	C.S.	Unk.
ESnet	465,135	52%	43%	3%
Internet2	18,774	1%	3%	13%
PlanetLab	718,459	56%	16%	1%
ShaperProbe	8,790	54%	9%	9%

agnoses. We did not find any false positive detections in manual inspection. We define "accuracy" as follows[11]. We ignore problems that were manually classified as "Unknown". We show two measures of diagnosis accuracy. For each problem:

- A diagnosis is correct if *at least one* of Pythia's diagnoses exists in the manually classified diagnoses. The diagnosis accuracy across all problems is 96%.
- A diagnosis is correct if *all* of Pythia's diagnoses exist in the manually classified diagnoses. The diagnosis accuracy across all problems is 74%. There were 1.42 diagnoses per problem on average.

Figure 4 shows validation accuracy for each pathology; we ignore pathologies for which we have less than 35 instances. A diagnosis for a problem is marked as "correct" if it exists in the ground truth. The "CongestionOverload" and "CongestionBursty" diagnoses include short-term delay level shifts; these are false diagnoses, since Pythia does not support them, and they comprise 42% of the total false diagnoses of the two congestion types.

9 Case Studies of Networks

In this section, we use Pythia to study performance problems in the Internet. We use measurements from production monitoring deployments in backbone and edge networks (see §8 for details of the data).

9.1 Monitor-related problems

Before we look at network pathologies, we study end-host (monitor) pathologies across different monitoring infrastructures. Our data comes from a wide variety of monitor platforms: Linux-based dedicated servers (ESnet and Internet2), desktop and laptops running different OSes at homes (ShaperProbe), and virtual machines (PlanetLab). Table 2 shows the fraction of detected problems that are diagnosed as end-host pathologies. We show frequencies separately for the two forms of end-host pathologies: "EndHostNoise" (short wait periods) and "ContextSwitch" (long wait periods).

The table shows an interesting feature that validates known differences between monitor types. The Internet2 monitors are dedicated resources for measurement tools, while the ESnet monitors also run MySQL databases for indexing measurement streams. Hence, Internet2 data shows a smaller fraction of end-host-related pathologies than ESnet, since the OS environment is more likely to be busy in ESnet monitors. The PlanetLab environment is also likely to be busy, given that the resources are shared among virtual machines. ShaperProbe data comes from a userspace tool running on commodity OSes, and where the users run other processes such as web browsers.[12]

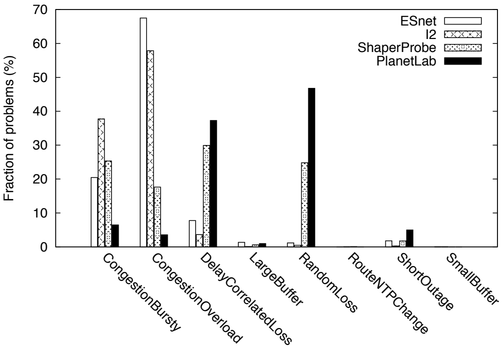


Figure 5: *Breakdown of network-based pathologies among the four datasets (omitting "Unknown" and end-host pathologies).*

The table also shows that the fraction of problems that Pythia diagnoses as "Unknown" are typically lower than 10%. In the rest of this section, we focus on network-related problems (i.e., excluding end-host and "Unknown" pathologies).

Implications: Production monitoring infrastructure is dedicated to measurements, and hence is not expected to induce large delays in measurements (other than monitor downtime). While this may be true for long-term monitoring (minutes to hours), we find that when the focus is on short-lived problems, we see a nontrivial proportion of monitor-induced delays. It hence becomes important to diagnose and separate such problems.

9.2 Core vs. Edge networks

We look at the composition of network-related pathologies in the different networks in Figure 5. We see that the high-capacity backbone networks, ESnet and Internet2, show a high incidence of congestion pathologies (both overload and bursty congestion). Moreover, there is no significant difference in the composition of pathologies between the two (similar) core networks.

The residential edge (ShaperProbe) shows a high incidence of both congestion and loss pathologies. Note that we do not see a significant fraction of "LargeBuffer" pathologies in home networks, though the presence of large buffers in home networks has been shown before

[11] Our diagnosis is a multiclass multilabel classifier that includes the "Unknown problem" output. It is not straightforward to define precision and recall in this case. We define accuracy as fraction of classified samples that are not unknown and are "correct".

[12] We note that "ContextSwitch" problems in ShaperProbe may include problems arising from either end-hosts or 802.11 links inside the home (Section 6.1). Separating end-host pathologies in ShaperProbe data allows us to focus on the ISP access link.

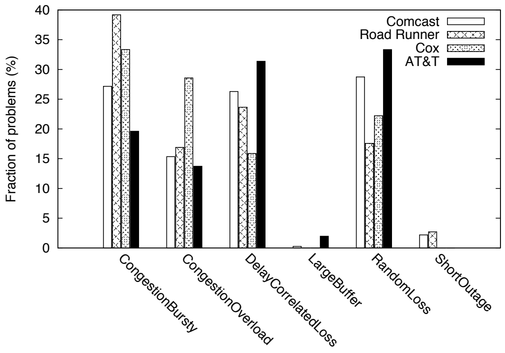


Figure 6: *Composition of different network-based pathologies among the four residential ISPs (both up and downstream).*

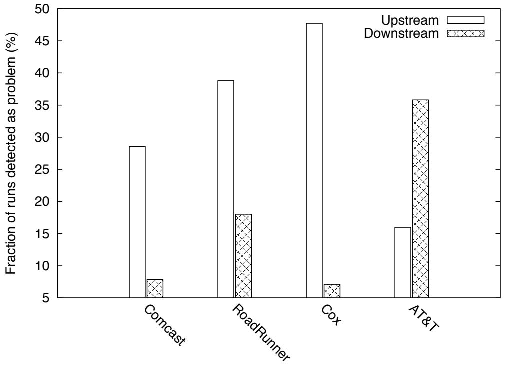


Figure 7: *Residential broadband networks: fraction of runs detected as pathology in upstream and downstream directions, for three cable and one DSL provider.*

[12]. This is because Pythia can diagnose a large buffer problem only if cross traffic creates a significant backlog in the buffer during the 10s probing time. PlanetLab data shows a high incidence of loss pathologies, but not congestion. The pathologies include delay-correlated and random losses, as well as short outages.

Implications: The results show differences in problems between backbone and edge networks. Despite the presence of large buffers, home networks are prone to loss-based problems, which can significantly degrade performance of web sessions. This may be due to 802.11 wireless links inside homes. A real time e2e diagnosis system can help quickly address customer trouble tickets.

9.3 Residential network pathologies

We look at four large residential broadband providers in our dataset: cable providers Comcast, Road Runner and Cox; and DSL provider AT&T. The number of network problems in our data depends on the number of runs ShaperProbe receives from ISP users (problems per link ranged from about 250 in AT&T to 12,000 in Comcast). Note that ShaperProbe data has an inherent "bias" – the tool is more likely to be run by a home user when the user perceives a performance problem.

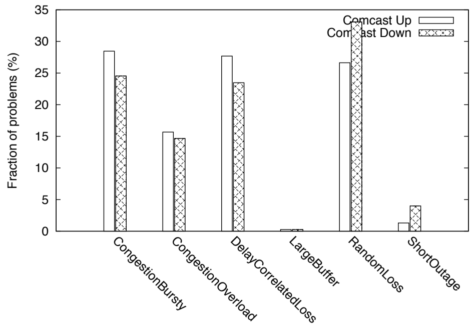


Figure 8: *Composition of different network-based pathologies in Comcast, as a function of link direction.*

Figure 7 shows fraction of user runs showing performance problem in the upstream and downstream directions for the four ISPs. We use runs which had recorded over 50% of expected number of measurement samples in 10s (at 10Hz). We see a difference in the frequency of problems in upstream and downstream directions between cable and DSL providers. A plausible explanation is that in the case of cable, DOCSIS uplink is a non-FIFO scheduler, while the downlink is multiplexed with neighborhood homes; DSL uses FIFO statistical multiplexing in both directions, but the link capacity is relatively more asymmetric. To cross-check the problem detection numbers, we look at the difference between 95th and 5th percentiles of the delay distribution during the problem. We find that about 59% upstream and 25% downstream Comcast runs have a difference exceeding 5ms (the default delay detection threshold); while for AT&T, the figures were 45% and 63% respectively.

Figure 6 shows composition of performance pathologies in each of the four ISPs. We see that the DSL provider AT&T shows a higher incidence of loss pathologies than the cable providers (both delay-correlated and random loss pathologies).

We next look at whether cable links show different pathology distributions among problems in the upstream and downstream directions. Figure 8 shows composition of pathologies across runs from Comcast. We do not see a significant difference in the composition of pathologies, even though there are more problem detections in the cable upstream than downstream.

Implications: We see that within residential ISPs the nature and composition of performance pathologies varies. Hence, we cannot build a one-size-fits-all system for residential ISPs; the system should allow operators to input domain knowledge-based pathologies.

10 Discussion and Limitations

In this paper, we presented Pythia, a system for detection and diagnosis of performance problems in wide area providers. Prior work has taken two key approaches to performance diagnosis in ISPs (see §11). The first approach involves designing and deploying *new probing tools* that diagnose specific pathologies. This enables detailed diagnosis, but new tools add probing overhead. The second approach involves *mining network-wide data*, from every network device and other data such as trouble tickets. This enables detailed diagnosis as well, but does not work in wide area ISPs, where paths can be inter-domain.

We explore a new complimentary approach that works with existing monitoring infrastructure and tools, and works in the inter-domain case. Pythia uses diagnosis rules, which are defined as logical expressions over problem symptoms. Operators can add these rules to the system using a specification language. This design enables ISPs to *incrementally deploy diagnosis* in production: not only when adding new monitors to the network, but also as and when *new* performance problems are seen by operators. At the same time, this approach eliminates the need for training data, which is hard to have in wide area ISPs. Pythia provides real time diagnosis summaries by using non-invasive detection and diagnosis algorithms at the monitors. In the course of building Pythia, we have noted some limitations and considerations when building and deploying the system.

Monitoring: Pythia relies on existing monitoring in the ISP. This could mean that the *diagnosis may be limited by probing* (e.g., probing frequency). For example, our results on home networks show a low incidence of large buffer-related problems, since not all large buffer delay increases may not sampled by the probing process.

Specifications: Language specifications of diagnosis rules may have limitations, despite the flexibility that they offer. When adding diagnosis rules, operators need to consider the tradeoff between specificity and generality of new diagnosis rules relative to existing rules – in particular with large number of rules. Further, it may not be feasible to specify some pathologies, since the monitored feature set, and hence the symptom set, is limited.

Sensitivity: In practice, Pythia's detection logic can lead to a large number of reported pathologies. We address this by including a knob that allows the operator to tune *sensitivity*, defined as the magnitude of deviation from the baseline (§4). We leave it to future work to rank pathologies based on operator interest and criticality.

Symptoms: The symptoms used in diagnosis rules may be based on models of performance (e.g., the end-host class), or may be based on static thresholds. We note that symptoms based on static thresholds may be common in practice, since they are likely to be based on operator experience. Since these thresholds could change with time, an open feature in Pythia is to extend the specification language to support threshold declarations.

Pathologies: Our deployment experience has shown that signatures induced by monitors may be common in practice when the focus of diagnosis is on short-lived problems. We leave open the analysis of problems that Pythia finds "Unknown". Pythia could augment these with a *similarity* measure over a specified space of features. Similarity compares the problem against a representative set of diagnoses to find the most similar diagnosis.

11 Related work

There has been significant prior work on detection and diagnosis of performance problems. The prior work falls into two classes of design. We present representative work in each class.

Data-oriented methods: These are diagnosis methods that use significant amount of data sources that are usually available in enterprises and single administrative domains, but *not* in wide area inter-domain settings. These methods give detailed diagnosis; there is a tradeoff, however, between how *detailed* the diagnosis can be and the wide-area applicability (*generality*) of a diagnosis method. A summary of some of these methods follows. AT&T's G-RCA [24] works on data from a single network such as SNMP traps, syslogs, alarms, router configurations, topology and end-to-end measurements. It mines dependency graphs from the data and constructs diagnosis *rules* from the graphs. SyslogDigest [19] mines faults from router syslogs. NetMedic [10] and Sherlock [3] diagnose faults in enterprise settings by profiling end-host variables and by mining dependencies from historic data. NICE [16] enables troubleshooting by analyzing correlations across logs, router data and loss measurements. META [23] looks at spatial and temporal features of network data to learn fault signatures. A recent study [22] uses email logs and network data to analyze routing-based failures. Learning methods may require prior training; however, they can complement Pythia by classifying problems that cannot be diagnosed using domain knowledge.

Data-oriented methods have also been used to diagnose specific pathologies in the context of a single network. Feather et al. look at diagnosing soft failures in a LAN using domain knowledge on a pre-defined set of features [5]. We take a similar approach to diagnosis using domain knowledge, but in the more general wide area inter-domain context. Lakhina et al. used unsupervised clustering methods on packet-level features in packet traces to classify performance anomalies [14]. Huang et al. use structural properties of packet traces to detect performance problems in a LAN [7]. Huang et al. identified inter-domain routing anomalies using BGP updates [8]. There has been extensive work on structural methods to detect anomalies in volume data in an ISP; for example, the influential work by Lakhina et al. uses dimensionality reduction on volume data [13].

Active probing methods: These methods rely on active probing, and are typically based on domain knowledge. They can reveal detailed and accurate diagnosis, since they provide the choice of carefully crafting probing structures. It is, however, hard to widely deploy a new active probing tool in a large monitoring network, especially if some of the monitors are in other ASes. We summarize a few representative tools below. Netalyzr [12] probes to help a user with troubleshooting information. Tulip [15] diagnoses and localizes reordering, loss and congestion using a single end-point. PlanetSeer [26] uses a combination of active and passive methods to monitor path failures. Prior work designed probing tools for specific diagnoses such as Ethernet duplex mismatch [20] and buffering problems [17].

12 Conclusion

In this paper, we have designed a performance problem detection and diagnosis system for wide area ISPs, that works in conjunction with deployed monitoring infrastructure. Pythia only requires a lightweight agent process running on the monitors. We have designed efficient detection and diagnosis algorithms that enable such an agent without affecting measurements. Pythia provides an expressive language to the operator to specify performance pathology definitions based on domain knowledge. We have deployed Pythia in monitoring infrastructure in wide area ISPs, diagnosing over 300 inter-domain paths. We used Pythia to study performance pathologies in backbone and edge networks. Our experience with Pythia has shown that existing monitoring infrastructure in ISPs is a good starting point for building near real time wide area problem diagnosis systems, enabling incremental diagnosis deployment.

Acknowledgements: We thank Jason Zurawski from Internet2, and Joe Metzger, Brian Tierney, and Andrew Lake from ESnet for providing us with OWAMP data sets. We thank Warren Matthews for his work on Pythia deployments. We are grateful to the anonymous reviewers for their valuable comments. This research was supported by the U.S. Department of Energy under grant DE-FG02-10ER26021.

References

[1] *Deployments of Network Monitoring Software perfSONAR Hit 1,000*. `http://1.usa.gov/Qt94Nk`.

[2] *perfSONAR Deployment on ESnet*. Brian Tierney. Presented at AIMS Workshop, CAIDA. 2011.

[3] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM CCR*, volume 37, pages 13–24, 2007.

[4] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet reordering is not pathological network behavior. *IEEE/ACM ToN*, 7(6):789–798, Dec. 1999.

[5] F. Feather, D. Siewiorek, and R. Maxion. Fault detection in an Ethernet network using anomaly signature matching. In *ACM SIGCOMM CCR*, 1993.

[6] A. Hanemann, J. Boote, E. Boyd, J. Durand, L. Kudarimoti, R. Łapacz, D. Swany, S. Trocha, and J. Zurawski. PerfSONAR: A service oriented architecture for multi-domain network monitoring. *Service-Oriented Computing*, 2005.

[7] P. Huang, A. Feldmann, and W. Willinger. A non-intrusive, wavelet-based approach to detecting network performance problems. In *ACM SIGCOMM IMW*, 2001.

[8] Y. Huang, N. Feamster, A. Lakhina, and J. Xu. Diagnosing network disruptions with network-wide analysis. In *ACM SIGMETRICS PER*, volume 35, pages 61–72, 2007.

[9] A. Jayasumana, N. Piratla, T. Banka, A. Bare, and R. Whitner. Improved Packet Reordering Metrics (RFC 5236), June 2008.

[10] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl. Detailed diagnosis in enterprise networks. In *ACM SIGCOMM CCR*, volume 39, pages 243–254, 2009.

[11] P. Kanuparthy and C. Dovrolis. ShaperProbe: end-to-end detection of ISP traffic shaping using active methods. In *ACM SIGCOMM IMC*, 2011.

[12] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *ACM SIGCOMM IMC*, 2010.

[13] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *ACM SIGCOMM CCR*, volume 34, pages 219–230, 2004.

[14] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *ACM SIGCOMM CCR*, volume 35, pages 217–228, 2005.

[15] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *ACM SIGOPS OSR*, volume 37, pages 106–119, 2003.

[16] A. Mahimkar, J. Yates, Y. Zhang, A. Shaikh, J. Wang, Z. Ge, and C. Ee. Troubleshooting chronic conditions in large IP networks. In *ACM SIGCOMM CoNEXT*, 2008.

[17] M. Mathis, J. Heffner, P. O'Neil, and P. Siemsen. Pathdiag: automated TCP diagnosis. *PAM*, 2008.

[18] R. Miller, W. Matthews, and C. Dovrolis. Internet usage at elementary, middle and high schools: a first look at K-12 traffic from two US Georgia counties. In *PAM*, 2010.

[19] T. Qiu, Z. Ge, D. Pei, J. Wang, and J. Xu. What happened in my network: mining network events from router syslogs. In *ACM SIGCOMM IMC*, 2010.

[20] S. Shalunov and R. Carlson. Detecting duplex mismatch on Ethernet. *PAM*, 2005.

[21] B. Silverman. *Density Estimation for Statistics and Data Analysis*. Monographs on Statistics and Applied Probability. Taylor & Francis, 1986.

[22] D. Turner, K. Levchenko, A. Snoeren, and S. Savage. California fault lines: understanding the causes and impact of network failures. In *ACM SIGCOMM CCR*, volume 40, pages 315–326, 2010.

[23] T. Wang, M. Srivatsa, D. Agrawal, and L. Liu. Learning, indexing, and diagnosing network faults. In *ACM KDD*, 2009.

[24] H. Yan, L. Breslau, Z. Ge, D. Massey, D. Pei, and J. Yates. G-RCA: a generic root cause analysis platform for service quality management in large IP networks. In *ACM SIGCOMM CoNEXT*, 2010.

[25] S. Zarifzadeh, G. Madhwaraj, and C. Dovrolis. Range tomography: Combining the practicality of boolean tomography with the resolution of analogue tomography. In *ACM SIGCOMM IMC*, 2012.

[26] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. USENIX OSDI, 2004.

BISmark: A Testbed for Deploying Measurements and Applications in Broadband Access Networks

Srikanth Sundaresan,[*] Sam Burnett,[*] Nick Feamster
School of Computer Science, Georgia Tech

Walter de Donato
University of Napoli "Federico II"

http://projectbismark.net

Abstract

BISmark (Broadband Internet Service Benchmark) is a deployment of home routers running custom software, and backend infrastructure to manage experiments and collect measurements. The project began in 2010 as an attempt to better understand the characteristics of broadband access networks. We have since deployed BISmark routers in hundreds of home networks in about thirty countries. BISmark is currently used and shared by researchers at nine institutions, including commercial Internet service providers, and has enabled studies of access link performance, network connectivity, Web page load times, and user behavior and activity. Research using BISmark and its data has informed both technical and policy research. This paper describes and revisits design choices we made during the platform's evolution and lessons we have learned from the deployment effort thus far. We also explain how BISmark enables experimentation, and our efforts to make it available to the networking community. We encourage researchers to contact us if they are interested in running experiments on BISmark.

1 Introduction

A defining feature of today's Internet is the proliferation of high-speed broadband access. The United States alone has more than 245 million broadband users, and usage statistics in other regions are even more impressive: at the end of 2012, China reported more than 560 million Internet users, with a penetration rate of more than 40% [20, 21], and Africa is seeing increased penetration and plummeting costs for high-speed connectivity [22, 28].

These changes encouraged us to study the nature and evolution of Internet connectivity as many users now experience it. We aimed to deploy a platform that could support continuous measurements from long-running vantage points and allow researchers to develop, test, and deploy new systems and services for common access network environments. To support rich and accurate Internet measurements from vantage points that are characteristic of typical Internet users, researchers need a testbed that represents the perspective of the growing population of Internet users.

Our vision for such a testbed was perhaps most comparable to long-running testbeds with dedicated hardware, such as PlanetLab [2] and the RIPE Atlas Project [33]. Inspired by these projects, we decided that deploying dedicated hardware, in the form of commodity home routers, was the best way to ensure that we could perform both long-running and on-demand measurements from a consistent set of vantage points where researchers previously did not have access. Such a deployment enables measurements that are *continuous* (unlike many measurement tools, which report only a single set of measurements initiated by the user), *direct* (unlike browser or host-based measurement tools, which can often reflect the performance of the host or application rather than of the network itself) and *comprehensive* (unlike client hardware, which cannot directly measure many aspects of both home and access networks). In contrast to PlanetLab, however, our goal was to focus on broadband access networks, as opposed to research networks, thus achieving a more *diverse* set of vantage points. Moreover, in contrast to the RIPE Atlas testbed, we designed the testbed to be *extensible*, supporting custom measurements, systems, and services. We also designed the testbed with user security and privacy as a first-order concern.

To address this need, we developed BISmark, a system that allows researchers, operators, and policymakers to deploy experiments and applications that gather data about network availability, reachability, topology, security, and performance from home routers running in globally distributed access networks.

Beyond the conventional challenges of operating a long-running service in the wide-area Internet (*e.g.*, PlanetLab), we faced a unique set of challenges when deploying such a testbed in *home networks*. First, incentives do not naturally align: whereas in PlanetLab, researchers have an incentive to host machines to gain access to the testbed, BISmark explicitly targets home users, who may not necessarily be interested in networking research. Second, unlike in universities where PlanetLab nodes are deployed, technical support is not readily available, which makes system robustness, remote maintenance, and recovery even more important. Third, nodes must be small and easy-to-deploy;

[*] These authors contributed equally to this paper.

such nodes are typically resource-constrained. Finally, BISmark nodes are on the direct path of real Internet users; a malfunctioning BISmark device not only disrupts a normal user's Internet connectivity but also typically results in the loss of the device. (At this point, a user is likely to remove the device from the network entirely and never re-install it.) Therefore, BISmark must also ensure that an unstable device or a poorly designed experiment cannot wreak havoc on the user's Internet connection.

BISmark has enjoyed reasonable success in its first four years. It has enabled the publication of many studies from broadband access networks from around the world, and is now being adopted by major ISPs, policymakers, and researchers in several countries. Many research groups are either using the data we have collected or deploying custom experiments. Yet, enabling a broader set of experiments and scaling BISmark beyond its current size poses new challenges. Security and robustness remain paramount, and device deployment and attrition are an uphill battle, particularly in certain regions. This paper discusses the constraints we have faced (and continue to face) in the design, implementation, and deployment of BISmark, discusses lessons and things that we would have done differently (or will change in the future), and describes new challenges as the platform expands both in terms of the number of vantage points and the diversity of experiments we aim to support.

The two audiences who will find this paper most useful are (1) designers and developers of network testbeds, who can read about BISmark's architecture and deployment lessons in Sections 3 and 5, respectively; and (2) researchers who want to use BISmark to collect measurements for their own work, who can read about how other researchers have used BISmark in Section 4. Anyone who is interested in deploying user-facing testbeds or measurement systems in hardware or software may learn from our experiences. More generally, we hope that anyone who has ever grappled with building a testbed—or plans to do so in the future—will take important lessons from our experience, many of which surprised us, and still others that may seem obvious in hindsight but are nonetheless well worth codifying.

2 Related Work

Fixed server or gateway deployments. PlanetLab [31] is probably the most similar platform to BISmark in that it aims to be a fixed, large-scale deployment hosting a variety of research experiments. Because BISmark is deployed in home networks on resource-constrained devices, however, it faces additional challenges. The RIPE Atlas [33] project has deployed thousands of probing devices worldwide, but their capabilities are limited to simple measurements (*e.g.*, ping, traceroute). SamKnows [35] has deployed thousands of home routers in the US and the UK, but only supports limited performance measurements.

Host-based deployments. Dasu [36] is a host-based software client. It has a very large footprint (tens of thousands), and allows a variety of network measurements from end hosts. However, its advantages of scale comes at the cost of decreased flexibility: (1) it lacks the permissions to run certain measurements due to application restrictions, (2) it cannot run continuous measurements (*i.e.*, since hosts can be turned off, moved, etc.), and, (3) the measurements can reflect limitations of the host or the application taking the measurement and thus do not reflect the performance of a fixed network vantage point. The Grenouille project in France [17] measures the performance of access links using an agent that runs from an end host inside the home network. Neti@Home [38] and BSense [3] also use this approach, but with fewer users than Grenouille. Network Diagnostic Tool (NDT) [9] and Network Path and Application Diagnostics (NPAD) [25] send active probes to detect issues with client performance. Glasnost [16] performs active measurements to determine whether a user's ISP blocks BitTorrent traffic.

Netalzyr [23] lets users conduct a series of tests from a browser, but measurements are not continuous, and researchers cannot run custom tests from a set of hosts—the measurements collected are fixed, and the set of hosts from which measurements are collected depend on the users who decide to run the tool.

Programming frameworks. The process of vetting BISmark experiments is manual (as it was in previous testbeds such as RON [1]), which will be a limiting factor as the deployment grows. BISmark must ultimately strike a balance between flexibility (allowing researchers to specify experiments) and a constrained programming environment (limiting researchers from specifying experiments that could interfere with home users). Previous work on sandboxed, programmable measurement environments, such as Seattle [8] or ScriptRoute [39], could ultimately serve as a useful environment for specifying BISmark tests.

Other measurement studies of broadband access networks. Previous work characterizes access networks using passive traffic measurements from DSL provider networks in Japan [11], France [37], and Europe [24]. Siekkinen *et al.* [37] show that applications (*e.g.*, peer-to-peer file sharing applications) often rate limit themselves, so performance observed through passive traffic analysis may reflect application rate limiting instead of the performance of the access link. Other studies have characterized access network performance by probing access links from servers in the wide area [12, 13].

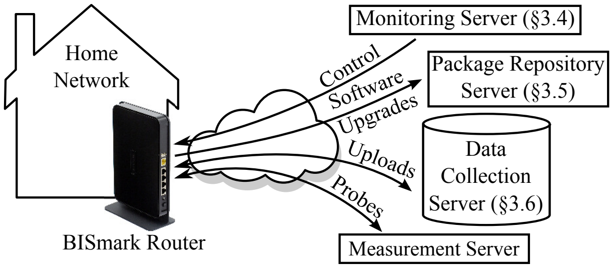


Figure 1: *BISmark architecture. The packages repository server and data collection server scale to multiple instances. The monitoring server is harder to scale, but also sees less load.*

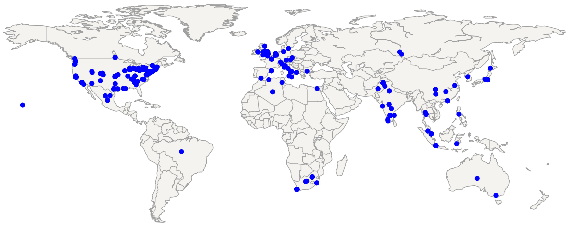


Figure 2: *Locations of the 178 BISmark routers that were online in January 2014. We have focused concentrations of routers in the US, South Africa, Pakistan, and the UK.*

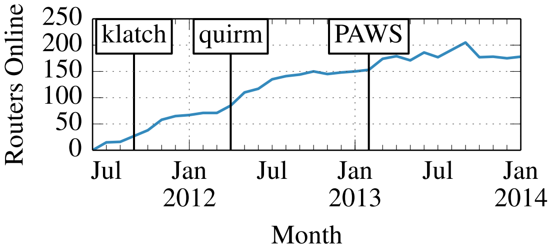


Figure 3: *The number of routers online during each month. BISmark has grown to nearly 200 routers over the past two and a half years.* klatch *and* quirm *signposts indicate two firmware releases;* PAWS *indicates a new deployment in the UK. Growth falters in some later months because we focused deployment in developing countries, where router availability is inconsistent. Numbers dropped in late 2013 when the PAWS study ended.*

3 Architecture and Implementation

BISmark aims to enable research and experimentation under constraints inherent to home routers and networks. Many of the challenges that we faced are not unique to our deployment, but they are exacerbated by operating (1) in a resource-limited setting on home routers; (2) in a setting where downtime (or general interference with users' Internet connectivity) is not acceptable.

BISmark's software fulfills four roles. First, it uniquely identifies each router and correlates it with metadata useful for conducting networking research. Second, it manages software installation and upgrades, which lets us fix bugs, issue security patches, and deploy new experiments after we have mailed the routers to participants. Third, it provides experiments a common, easy-to-use, and efficient way to upload data to a central collection server. Finally, it enables flexible and efficient remote troubleshooting. We describe BISmark's evolution path, its components, and the various roles that the BISmark software plays.

3.1 System Components

Figure 1 shows BISmark's architecture. The deployment currently comprises BISmark routers and a collection of servers that manage software, collect data, and facilitate troubleshooting.

BISmark routers. As of January 2014, the deployment has 178 active routers in over 20 countries. Figure 2 maps router locations and Figure 3 graphs the deployment's growth over the past two and a half years. We purchased, prepared, and mailed nearly half of these routers, while the rest have arisen either organically (*e.g.*, as users "flash" their own routers with BISmark firmware) or through coordinated efforts (*e.g.*, with other organizations or research groups). We currently deploy Netgear WNDR 3800 routers, which have a 450 MHz MIPS processor, 16 MB of flash storage, 128 MB of RAM, 5 gigabit Ethernet ports, and a dual-band wireless interface. This hardware is limited, even when compared to other embedded mobile devices like smartphones, yet it is powerful enough to reliably support both basic routing features and a variety of measurement experiments.

We replace the router's default software with a custom version of OpenWrt Linux [29]. OpenWrt has an active developer community, a simple and usable configuration GUI, and broad, mature hardware support. It frees us from maintaining our own firmware, but ties us to its release cycles, bugs and all. Despite a few persistent problems that have occasionally affected some users, we have been satisfied with OpenWrt. We have deployed two hardware revisions of Netgear routers and four firmware releases.

Some users download the BISmark firmware from our project page and install it on their own hardware. This enables further growth, but introduces the challenge of determining the identities of these users. To handle this scenario, our latest firmware includes a user registration system that prompts users to complete a simple registration process after configuring network settings. Registration serves two purposes: it automates the previously manual collection of metadata from users about their ISP; and it binds the user's identity to a router so we can restore the router configuration each time the router is reflashed.

Section 5.4 discusses the security implications of letting users install BISmark on their own hardware.

Management servers. To support the router deployment, BISmark has three types of servers:

1. *Package repository servers* decide which software should run on each router. Different routers can run slightly different sets of software because we've deployed several firmware versions and users have consented to run various experiments.
2. *Data collection servers* validate, store and serve data gathered by routers. We serve publicly accessible active measurements data from Amazon S3 and mirror all data in servers at our university.
3. *Monitoring servers* track availability and can initiate SSH connections to routers for troubleshooting.

Measurement servers. BISmark uses a fixed set of measurement servers against which it conducts standard performance measurements (*e.g.*, throughput). The validity of these experiments in many cases depends on having measurement servers that are geographically close to the deployed routers. We have been fortunate to obtain access to the globally distributed Measurement Lab (MLab) infrastructure [26]. In some cases where we have a critical mass of routers, we have also deployed additional measurement servers. Measurements are scheduled on the measurement servers by a central scheduler. This prevents overloading of servers by several concurrent requests from BISmark routers. We note that this infrastructure is used for intensive active tests such as throughput measurements. Other experiments that do not rely on a low-latency, globally distributed infrastructure do not use these servers.

3.2 Architectural Constraints

Like many rapidly growing systems, BISmark evolved organically in response to use. Several components written for an early pilot deployment persist. Although many design choices were sub-optimal in retrospect, the software has always addressed three practical constraints.

Constraint 1: *Severely limited client resources dominate software design decisions.*

Resource limitations preclude several conveniences. For example, we cannot run heavy scripting languages like Python or Ruby; instead, we glue together standard UNIX utilities and small C programs with shell and Lua scripts.

Constraint 2: *The basic routing functionality of BISmark routers is critical; users often place them on the home network's forwarding path.*

The router should not noticeably affect the user's networking experience (*e.g.*, by frequently saturating the uplink). Combined with limited client resources, this constraint requires us to thoroughly test software before deploying it, because the consequences of malfunctioning software may be the potentially terminal loss of a deployment site. (In our experience, most users simply unplug the router at the first annoyance and never plug it in again.)

Constraint 3: *User intervention is impractical and should be as limited as possible.*

Users expect their router to "just work". They have no desire to otherwise interact with it. After installation, attempts to interact with users via the router itself are awkward and annoying (*e.g.*, captive portals) and out-of-band communication (*e.g.*, email) is unreliable.

3.3 Naming

We assign each router a unique *router identifier*, which we use for data analysis, troubleshooting, and inventory; we correlate this identifier with all measurements we collect from the router, participant-provided details about the upstream ISP's advertised performance, the router's geographic location, and the participant's name and mailing address (used to ship the router). We do not disclose personal information except when required by law enforcement [5] (a scenario that we have not yet encountered).

Constraint 4: *Router identifiers must be (1) unique and (2) persistent across reboots and reflashes.*

Common identifiers such as manually assigned hostnames, dynamically generated tokens, or public IP addresses do not satisfy these requirements. Instead, we use the routers' LAN-facing MAC address, which is both unique and unchanging. We chose LAN (rather than WAN) addresses because they are printed on the back of the router, which simplifies technical support and inventory.

Unfortunately, LAN-facing MAC addresses pose a security risk because attackers could use them to geographically locate a router. By default, routers broadcast their MAC address to WiFi devices in the vicinity, including smartphones and collectors for Google's Street View and similar data collection projects. An attacker with access to both the router's MAC address and these databases could geolocate a router [43]. This vulnerability highlights a broader set of tradeoffs BISmark makes between privacy and transparency; Section 5 discusses these tradeoffs.

3.4 Troubleshooting

Remote access allows us to fix critical problems that would otherwise require a lengthy packaging and software update cycle to fix.

Constraint 5: *We need a fast but secure technique to log in to routers on demand.*

Every BISmark router runs an SSH server which is only exposed on the LAN interface. Exposing the server on the WAN interface has security concerns; additionally, over 60% of our routers are obscured by at least one layer of Network Address Translation (NAT), rendering the

SSH server on WAN useless. Instead, BISmark routers poll the *monitoring server* for SSH session requests by sending small UDP probes ("heartbeats") once per minute. If the server wishes to initiate an SSH session with a router, it responds with a UDP response to that effect; the router then opens an SSH tunnel forwarding a port on the monitoring server to the router's local SSH server. Administrators on the server then initiate SSH sessions to the forwarded server port. Although session requests and responses may be lost in the network, we can usually establish a tunnel within one minute and almost always within five minutes.

Tunnel creation uses restricted SSH authorization to prevent routers from executing arbitrary commands on the server. Because the server does not authenticate the UDP probes themselves (only the resulting tunnels), we rate limit requests to prevent denial-of-service or reflection attacks against the server or unsuspecting routers.

The overhead of the tunneling protocol is minimal. Each probe/response pair is 139 bytes, resulting in an overhead of 200 KB per day or 6 MB per month. The DNS time-to-live on the monitoring server is 15 minutes, resulting in an additional overhead from DNS lookups of less than 1 MB per month. Using a domain name with a reasonable TTL allows us to quickly migrate the monitoring server in an emergency. Although a DNS hijacker could direct routers' probes to a different address, such an attack will not compromise security of the routers because they use SSH to establish tunnels.

3.5 Software Upgrades

After we have deployed a router, we must be able to manage its software packages. Throughout the lifetime of the deployment, we have issued many bug and security fixes as package upgrades, deployed new measurement experiments by installing new packages, and rolled back faulty experiments via package removal. OpenWrt's built-in *opkg* lacks several features and safeguards necessary for managing software on a large deployment in the homes of non-technical users. This section illustrates several of BISmark's unique software management constraints and we overcome them.

Constraint 6: *Router state should be centrally managed and controlled.*

Exogenous events can interfere with stateful package management. For example, if a user resets their router to its original configuration, the router should automatically install and upgrade the software it had prior to the reset. Instead of executing one-time commands sent by a server, the router downloads a list of packages reflecting the current desired state of the router, and installs, removes, and upgrades its packages accordingly.

We built custom package management utilities on top of *opkg* that meet these constraints. We eschewed off-the-shelf tools like Puppet, Chef, or CFEngine because of resource and complexity constraints, which reflects a general tension between custom and commodity software that we faced throughout the project. Existing software is often both more complicated than we need and untested on non-x86 architectures. Sometimes writing a small custom package from scratch for limited functionality is easier than porting and rigorously testing an existing one.

Constraint 7: *Software package management must occur without intervention from either home network users or the BISmark administrators.*

This constraint contrasts with large-scale software administration frameworks for other platforms (*e.g.*, Android, Mac OS X, and Windows), which have at least some user interaction. Home users are often non-technical or otherwise have little desire to administer their router, and the BISmark administrators cannot manually run package commands on hundreds of routers. Therefore, software installation, upgrade, and removal must happen automatically. Routers in our deployment automatically perform these tasks at boot and approximately every twelve hours.

Constraint 8: *The consequences of accidental installation, upgrade, or removal of packages are high.*

Automating package management increases risk, because a single faulty or buggy package could cripple the entire deployment. Our management tools impose several restrictions to guard against accidental installation, upgrade, and removal of packages. First, routers only allow removal of packages that are not included with their base firmware image, which prevents us from accidentally removing critical software. For example, before instituting this restriction an errant experiment accidentally removed the `libc` package from a router during testing, rendering it unbootable. Second, packages must be explicitly tagged as "upgradable" on the server before routers will upgrade them, which helps prevent us from accidentally pushing newer, incompatible versions of packages. Finally, we require all new packages and versions be tested for several days on a small set of *canary routers* owned by members of our research group before wider deployment.

Routers usually sit on a home network's critical path and must continue functioning at all times. Fortunately, basic function only relies on a few core packages, and almost always continues to work even if our management tools fail. Even if we lose access to the router entirely, the home router's core functions are typically undisturbed, since those functions are isolated from our packages.

Constraint 9: *Routers have diverse packages and versions.*

There are three versions of the BISmark firmware currently deployed; each requires a different set of packages because of library incompatibilities. Some routers also run additional packages because their users participate in optional BISmark measurement experiments. Our package management tools must install the correct packages and versions on each router. The consequences of mistakes range from broken functionality (*e.g.*, missing measurements) to unauthorized collection of data (*e.g.*, if we collect passive measurements from users who have not consented to it.)

3.6 Data Collection

Experiments generate data of a wide variety of sizes and rates and upload it to data collection servers for analysis. We initially used off-the-shelf file synchronization tools to upload this data, but resource, flexibility, and reliability constraints motivated us to develop custom software. *bismark-data-transmit* is our client application, which detects new files in a filesystem subtree using Linux's *inotify* filesystem monitoring interface. The client uploads these files over HTTPS to a Web application that validates and archives the data.

Constraint 10: *Bandwidth is scarce and may be capped.*

Efficient use of bandwidth is crucial because BISmark routers share uplink capacity with the home network. Experiments can generate lots of small files, often frequently, but need to upload them with minimal protocol overhead. Routers store these files on a volatile RAM disk and upload files as soon as possible; both excessive wear and extreme scarcity prevent them from storing data on persistent flash storage. To minimize the risk of data loss when they lose power or reboot, routers do not batch files for more efficient transmission. To avoid frequent and expensive handshakes while still continuously uploading files, we send all files over a single HTTPS connection with a high keep-alive timeout. Although Web services generally use keep-alive timeouts of only a few seconds, *bismark-data-transmit* sets a one hour timeout, which makes sense for communicating with a small, fixed set of clients.

Constraint 11: *Network and power are unreliable in some locations; data collection should gracefully handle uplink outages and power outages.*

Some experiments measure properties of the home network itself rather than the home's Internet connectivity and thus continue generating data even when the router's uplink is offline. These experiments can quickly accumulate a lot of data if unchecked. *bismark-data-transmit* retries failed uploads every three minutes and starts discarding data in FIFO order once it has more than 24 MB queued for upload, so even routers connected to very unreliable uplinks can contribute data without exhausting their limited onboard storage. Some routers frequently lose power, particularly those deployed in developing countries. *bismark-data-transmit* minimizes data loss in these cases by uploading data as soon as possible. Routers can lose data accumulated during a long uplink outage that immediately precedes a power outage.

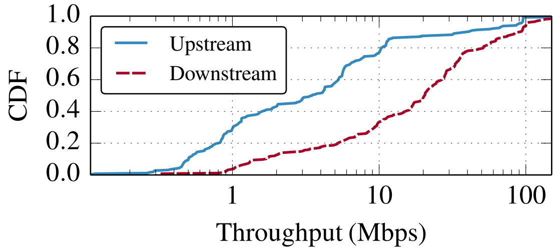


Figure 4: *Downstream and upstream throughput for routers.*

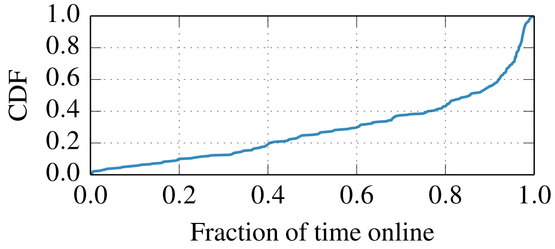


Figure 5: *Distribution of the fraction of time each router is online during its deployment. We only include routers that have been online for at least a month.*

4 Experimentation on BISmark

This section describes research projects that have used BISmark. We first describe the modes of collaboration that we have used since opening BISmark to external researchers in mid-2013. Because the platform is both resource constrained and on many users' critical path to the Internet, experiments on BISmark must cope with stricter conditions than most existing testbeds that support long-running deployments (*e.g.*, PlanetLab). For example, experiments must deal with nodes that have highly variable connectivity and availability. Figure 4 plots the 95^{th} percentile of throughput of homes in the deployment; we see ranges from basic broadband (about 1 Mbps) to fiber speeds (100 Mbps). Figure 5 shows the fraction of time a router is available and online during its lifetime; about 50% of the routers are available more than 90% of the time, but a significant fraction of routers have much patchier availability.

4.1 Modes of Collaboration

Data from many active measurements are public for anyone to use. Additionally, we have been advertising BISmark to collaborators and encouraging them to run experiments on the deployment. Most of this recruiting has been through word-of-mouth, as we build confidence that we can support a larger group of researchers (in fact, we anticipate that we would be unlikely to satisfy all requests). In many of these cases, we have informally adopted a PlanetLab-like incentives model by asking the researchers to spearhead their own small deployment of BISmark routers in an ISP or region of interest. In certain research projects, the researchers want to do this anyhow because they have a specific group of users that they want to study. We have two modes of collaboration, which we outline below. In both cases, researchers must be comfortable with OpenWrt and embedded platform development.

Public deployment. Collaborators run experiments on the main deployment of routers, which we manage. We control access and schedule the experiment to run in conjunction with other experiments running on the deployment. This mode works well for researchers running lightweight experiments from the variety of vantage points that our deployment offers. We have enabled research from the University of Southern California in this way.

Private deployments. Researchers (or, in some cases, operators) purchase and deploy their own routers, while we provide the client software and manage back-end services. In these cases, the researchers retain a high degree of access to their routers, giving them an incentive to keep their deployment running. This mode is best for researchers who want to run complex or time-consuming experiments in a small geographic area. For example, University of Cambridge has deployed more than 20 BISmark routers in under-privileged communities in Nottingham to study the mechanics of Internet sharing in such communities. We have also engaged with several ISPs who have wanted to run their own autonomous deployments.

4.2 Research Projects

BISmark offers the ability to study poorly understood or understudied aspects of home networks, including access link performance, application characterization, user behavior patterns, security, and wireless performance. Table 1 summarizes several experiments we are coordinating on the deployment. In many cases, we are leading (or have led) the study ourselves; more recently, we have been collaborating with the researchers who are leading the study. The latter projects are works-in-progress. The following sections describe both sets of projects in more detail. Our discussion of experiments that have been run on BISmark is not exhaustive but is intended to highlight both the capabilities and shortcomings of the platform.

4.2.1 Performance Characterization

BISmark's location at the hub of the home network lets it gauge performance of both local devices and the access link without being affected by confounding factors from the rest of the network.

Broadband performance in the US and abroad. The home access point is ideally suited for measuring access link characteristics. We characterized access link performance and the effects of access technology and customer premise equipment in the United States [40] using data from BISmark and the similar FCC/SamKnows deployment. Our study showed, amongst other things, how the access link can have a significant impact on end-to-end performance. Research ICT Africa (RIA) reproduced our study in South Africa [10] and expanded it to include mobile devices and 3G dongles. BISmark is well suited for such studies because of its view into the last mile, and its ability to account for confounding factors and to run longitudinal experiments.

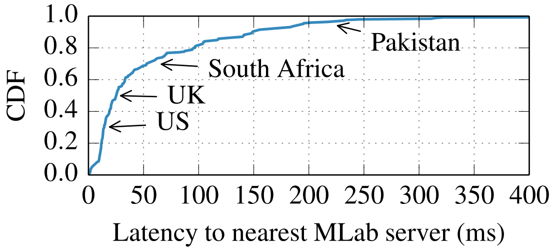


Figure 6: *Distribution of latencies to the nearest MLab server from each BISmark router, with annotations of the median latency from several countries with many routers. Latencies from Pakistan are very high because the nearest server is in Europe.*

Application performance. Because hardware limitations can prevent us from running full applications (*e.g.*, Web browsers), we aim to emulate applications' network behavior. In our work measuring network bottlenecks in Web performance [41], we used a custom browser emulator to measure one aspect of Web performance—the impact of the last mile. This study found that Web performance becomes bottlenecked on latency for broadband connections faster than 16 Mbps. Although BISmark was suitable for this particular experiment, we did not measure other aspects of Web performance, such as user perception, the effect of object ordering, or scripting on performance. This is because we cannot run a full browser on the router; it is also difficult to get such information from passive data. In such cases, we have to carefully design the experiment so that we know what we are able to study.

Wireless performance. We are developing techniques that isolate the source of performance bottlenecks to either the access link or the wireless network, as well as tools that help us understand the nature of wireless pathologies. The home access point sits between two common sources of performance issues—the access link and the wireless network—and is therefore ideally suited for identifying and isolating problems between these locations.

Lessons and caveats. As demonstrated above, BISmark is ideally suited for access link and home network characterization because it lets us probe these components and

Project	Institution(s)	Description	Publications
		Performance Characterization	
Broadband performance	Georgia Tech, University of Napoli, INRIA, FCC/SamKnows, Research ICT Africa, National University of Sciences and Technology	Study factors affecting broadband performance in the US and in developing countries.	[10, 40], WiP
Web performance	Georgia Tech, INRIA	Characterize and mitigate last-mile bottlenecks affecting Web performance.	[41]
Home wireless performance	Georgia Tech	Study home wireless pathologies and bottlenecks in home networks	WiP
		Usage and Home Network Characterization	
Home network characterization	Georgia Tech	Understand usage and connectivity.	[18]
Home Constant Guard	Comcast	Expand Constant Guard to provide information about devices infected in home networks.	WiP
PAWS	University of Cambridge	Internet sharing in underserved communities.	WiP
		Topology and Connectivity Characterization	
Google cache measurements	University of Southern California	Study effects of Google's cache deployment on performance of Web services.	[7], WiP
Network Connectivity	Georgia Tech, USC, RIA	Characterize ISP connectivity and path inflation in Africa.	[19]
Network outages and DHCP	University of Maryland	Study effects of outages on IP address allocation worldwide.	WiP
OONI/censorship	NUST, University of Napoli	Study the extent and practice of censorship in various countries (initial focus on Pakistan).	WiP

Table 1: *Summary of various experiments (and publications) that BISmark has enabled to date. "WiP" denotes work in progress.*

collect passive data from them. Application performance characterization is harder. Applications (or their emulations) must be light enough to run on the router; this might preclude certain types of applications.

Experiments that measure the access link by sending active probe traffic (*e.g.*, throughput tests) must not degrade performance of the home network while doing so. For users with bandwidth caps, probe traffic and data traffic (from uploading measurements to the server) should not constitute a significant fraction of the cap without the user's knowledge or consent. Some measurements such as TCP throughput require server deployments with low latency. Fortunately, Measurement Lab's global server infrastructure has helped: BISmark nodes automatically select the nearest MLab node for throughput measurements; Figure 6 shows that over 80% of nodes are within 100 ms of a measurement server.

4.2.2 Usage and Home Network Characterization

Several projects leverage BISmark's view of the home network behind the NAT.

Home network availability, usage, and infrastructure. We study the kinds of devices home users use to access the network, how they access the network, and their usage patterns [18]. Our study found interesting behavioral patterns (*e.g.*, users in developing countries turn off home routers when not using them) and usage patterns (*e.g.*, most traffic is exchanged with a few domains). Most prior studies are incomplete because they rely on one-time or infrequent probes from clients with limited network visibility.

Home network security. The ability to isolate traffic from different devices behind the NAT can be used to improve security. Comcast offers a security solution called Constant Guard, which captures DNS lookups to suspicious domains to inform a user when devices in their home may be compromised. We are extending BISmark to let Constant Guard identify infected devices and redirect some or all flows from suspected devices through Comcast security middleboxes via a virtual private network [4].

Internet usage in underprivileged communities. The PAWS project [30] distributes BISmark routers augmented with extra measurement tools to broadband customers who volunteer to share their high-speed broadband Internet connection for free with fellow citizens. The project studies how underprivileged communities share Internet access.

Lessons and caveats. BISmark's view into the home network and its ease of deployability allows it to run experiments that are not possible with other platforms. However, it also raises new concerns. Resource constraints limit the amount of data that can be collected and processed on the device. User privacy is a significant concern; for any experiment that studies user behavior, we must obtain informed consent from the user, which can be a slow and cumbersome process. We have conducted our own experiments that have required institutional review board (IRB) approval, but complications arise when BISmark serves as a host platform for experiments run from other universities that are sometimes in other countries. Even with permission to collect personal information, we design our experiments to collect only information necessary to answer targeted questions.

4.2.3 Connectivity Characterization

BISmark's worldwide presence lends itself to measuring many different aspects of Internet connectivity.

Measuring Internet topology and connectivity. We have looked at Internet availability in developed and developing countries [18], and researchers at USC are using BISmark to extend their study of the effects of Google's expanding cache deployment [7] on the performance of various Web services. Researchers at the University of Maryland are analyzing BISmark's UDP "heartbeat" logs (Section 3.4) to understand the effects of network outages on DHCP address allocations. Our recent work explores correlated latency spikes in ISPs [34] and the extent to which interconnectivity (or lack thereof) at Internet exchange points contributes to latency inflation and degraded application performance [19].

Global measurements of censorship. BISmark routers represent a unique opportunity to collect detailed, longitudinal data about how countries engage in censorship. Researchers in Pakistan have deployed BISmark routers in several homes to measure this phenomena; routers in other countries could also potentially collect similar measurements. We are replicating OONI [14] on BISmark.

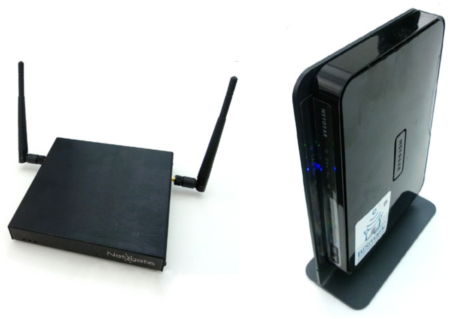
Lessons and caveats. BISmark is well-suited for connectivity measurements because of its geographical footprint, availability, and its ability to run periodic measurements (time scale of minutes) over a long period of time (months, or even years). We could also run probes and tests between BISmark routers, but while some such experiments may be better run on platforms like Dasu which will likely always have more deployment sites, BISmark is a full-featured (if low-powered) Linux machine that offers the ability to perform a much larger variety of experiments. Experiments that measure censorship have additional ethical concerns because it is illegal in some countries, and may even place the household at risk. In these cases, we must obtain informed consent, which may be difficult with users who flashed their own hardware or don't speak English.

5 Lessons

This section summarizes lessons we have learned (often the hard way) during BISmark's development.

5.1 Recruiting Users

Convincing users to deploy routers in their homes, particularly custom hardware, is not easy. Prior to our current deployment of commodity routers, we conducted a year-long pilot study with the NOX Box, a small form-factor PC that ran the NOX OpenFlow controller on Debian Linux. We assembled the hardware from an ALIX 2D13 6-inch by 6-inch board with a 500 MHz AMD Geode processor, 256 MB of RAM, 2 GB of flash memory, three Ethernet ports, and a wireless card. Although the NOX Box's relatively unconstrained hardware and full-featured Linux distribution helped rapid prototyping, our pilot faced several practical problems in the field.



(a) *NOX Box* **(b)** *Netgear WNDR 3800*

Figure 7: *We used the NOX Box for our pilot deployment and the Netgear WNDR3700/WNDR3800 for the second deployment. Unlike the NOX Box, the Netgear router looks like standard home networking equipment.*

Lesson 1: *Form factor matters. Users often trust commodity hardware over custom hardware simply because it is in a recognizable form.*

Figure 7 compares the NOX Box hardware from our pilot phase to the commodity Netgear hardware from our current deployment. The NOX Box doesn't look like a typical home router: it lacks familiar branding, lacks labels for status lights and Ethernet ports, and has a metal rather than plastic enclosure. These factors bred an inherent distrust of the NOX Box. People were generally more willing to deploy commodity hardware.

Lesson 2: *Users often blame BISmark for problems in their home network, whether or not the problem was caused by BISmark. Many users react by removing the router permanently from their network.*

Even with commodity hardware, users have heightened awareness of the BISmark router, particularly the experimental nature of the device, and therefore suspect it first when problems arise with their home network. In some cases, BISmark is indeed at fault. For example, a firmware bug causes unstable wireless connectivity on some devices, notably Apple MacBooks. In other cases, the router uncovered buffering problems elsewhere in the home network, temporarily degrading network conditions in the process. Many times, users misconfigured the router themselves (*e.g.*, by changing firewall settings) or incorrectly blamed BISmark for upstream ISP outages or problems with end hosts (*e.g.*, older devices that lack support for WPA2.)

Regardless of the causes of these problems, many users "solve" them by removing the BISmark router from the network. This has consequences in terms of money (the router likely will not be turned on again) and time (in flashing, packaging, and shipping the router to the user).

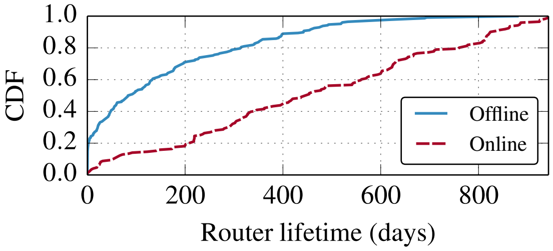


Figure 8: *Distribution of router lifetime, the difference between the first and last time we saw the router online. For 178 routers currently online, it is how long they have been online to date. 169 are now offline, while 31 never turned on at all.*

Lesson 3: *Home users and researchers have vastly divergent incentives. Home users want a working network, and researchers want to gather data and information. Care and effort must be invested to align these incentives.*

It is critical that our deployment strategy allows us to finance and maintain a large number of routers. PlanetLab's incentive structure (*i.e.*, hosting infrastructure for the right to run deployment-wide experiments) does not work in our case, because many of the most interesting vantage points will be hosted by users who are not networking researchers and have no interest in conducting their own experiments. We use two deployment strategies:

Free (or subsidized) router distribution. Our initial strategy was to ship routers to acquaintances, friends of friends, and through targeted advertising in venues such as NANOG and Internet2. It is difficult and time consuming to track routers in such cases, particularly when users turn them off. Due to the cost and effort involved, individual shipments only work at relatively small scales. About 50% of routers we distributed have either never been turned on or have since been decommissioned by their users.

Federated distribution. We are now attempting a federated deployment model to expand our geographical footprint. We work with a local contact who buys or receives a shipment of routers from us, recruits volunteers and ensures that routers stay up. This approach worked well for a deployment of approximately 15 routers in South Africa and 20 routers in the UK, and we are now attempting similar approaches in Tunisia, Pakistan, Nigeria, Cyprus, and Italy. We are also working with Comcast to deploy routers in their customers' homes in return for network analytics.

5.2 Sustaining the Deployment

Even after deploying BISmark routers in homes, it is a struggle to keep them online. Figure 8 shows attrition of the deployment. Nearly 25% of all routers go offline within three months, while another 25% have remained online for more than a year. We have learned many lessons, both about how to deploy reliable router software and how to keep users involved when unreliable software disrupts the user experience.

Lesson 4: *Users must be engaged to help keep routers online. Engagement can come in a variety of forms, and may be as simple as helping them better understand their network using the data we collect.*

If users disconnect their routers, we stand to lose both the device hardware and the data. We attempt to keep users engaged by providing useful tools like the Network Dashboard [27] to visualize ISP performance and uCap [42] to help users visualize and manage their home network usage. We conduct our development and data collection in the open; users can track BISmark development online [6, 32].

Lesson 5: *Upgrading critical software in the field is risky, but the ability to upgrade other software is essential for sustainable deployment.*

The ability to upgrade non-critical software after deployment has enabled us to pursue "good-enough" software development by deploying systems that are not fully ready. We can fix bugs and deploy new features by upgrading software in the field. This helps us reconcile the need for deploying systems that work for end users with constraints that include the need for extensibility and experimentation and limited time and manpower for testing and support.

We do not update certain critical software when there is a chance, however small, that a critical functionality (*e.g.*, the wireless network) could break. Our approach has been to minimize such cases by using a well-tested base platform that can maintain basic functionality even when higher-level client and backend software malfunctions.

Lesson 6: *Every home network has unique conditions and usage patterns, making comprehensive testing before deployment nearly impossible; bugs arise in practice.*

We aim to ensure that BISmark is foremost a stable access point, and that our custom software and experiments do not degrade user experience. The BISmark gateway is on the critical path of Internet access for at least one device in 92% of the homes; a malfunctioning router will disrupt network connectivity and, in the worst case, even completely take those devices offline. We have one window of opportunity per user to ensure that a router is installed and working correctly. Most people have no desire to troubleshoot home networks and will readily disconnect their BISmark router if it stops working as intended.

Lesson 7: *Community support is crucial; we rely heavily on commodity hardware manufacturers and open source software developers to build reliable, usable home routers.*

Commodity hardware solves many problems we faced with custom hardware because manufacturers (*e.g.*, Netgear) design hardware specifically to deploy it in the homes of non-technical users, which exactly matches BISmark's deployment scenario. Besides appearance, commodity hardware addresses many of the NOX Box's reliability and usability problems: (a) flash storage cards failed after only 3–4 months in the field, far sooner than expected for our workload; (b) we assembled each NOX Box from components, a laborious process; and (c) the component cost for each device was approximately $250 USD, or 2–3 times the cost of a commodity wireless router.

Similarly, OpenWrt's global community ensures that it is much more comprehensively tested on a variety of home routers than Debian. Although we occasionally found ourselves "held hostage" by bugs that were not fixed on our timescales, becoming embedded in OpenWrt's developer community was generally helpful.

Lesson 8: *Users may want to customize router settings, but doing so may introduce security vulnerabilities.*

BISmark routers have a flexible administrative interface to help users configure the router; this is a potential security vulnerability. One household accidentally disabled the router's firewall, opening its DNS resolver to the Internet. Attackers eventually recruited the router for amplification attacks over a period of many months, which we only discovered when the ISP notified the user of the problem. Although the disabled firewall was the culprit in this case, it led to a wholesale audit of the deployment and a spirited email exchange with the affected user. It is still unclear exactly how the firewall was disabled.

5.3 Experimentation

Designing and deploying measurements on BISmark has highlighted several nuances of supporting experimentation in production home networks.

Lesson 9: *It is difficult to reconcile the need for open data with that of user privacy.*

To encourage open data, we publish measurements collected from BISmark, but only if doing so doesn't threaten user privacy. Sometimes this decision isn't obvious. It is unclear when active data measurements can yield insight into user behavior; for example, patterns in router availability or throughput and latency measurements could indicate when users are home and using the network.

Lesson 10: *Vetting experiments is challenging, and a poorly designed (or controlled) experiment can cripple a user's Internet connection.*

Enabling a wide range of experiments introduces management and security concerns, specifically with reviewing code, controlling access, and ensuring that experiments do not disrupt user experience by making the device unstable or consuming too much network resources.

One household had comparatively slow upstream connectivity (512 Kbps upload) and an old modem with a large buffer, where even short throughput tests can induce bufferbloat pathologies [15]. Although the household's typical workload did not stress the network often enough to expose bufferbloat in their typical usage, BISmark's periodic throughput tests saturated the buffer and rendered the Internet unusable for the duration of the test (a few seconds). The degradation was bad enough for the user to complain and stop using the device after a few weeks.

5.4 Security

BISmark routers should compromise neither home network security nor the integrity of the platform. Although we try to minimize the possibility of security vulnerabilities by adopting industry-standard software and protocols wherever possible, some attacks against BISmark's backend infrastructure are still possible.

Lesson 11: *Users have access to the hardware and can modify firmware; this imposes new security challenges.*

BISmark's backend is subject to two broad security threats. The first is denial-of-service attacks, where malicious users could attempt to exhaust server resources for processing legitimate routers or measurements. Attackers could impersonate other users or even mount Sybil attacks to create many fake routers. Several backend components employ rate limiting, but these limits generally only protect against errant behavior of non-malicious clients. Thus far, we have deliberately chosen *not* to fix this class of vulnerabilities to make it easier for people to install BISmark on their routers without our involvement.

Other attacks could contribute malicious data to influence conclusions. Mitigating such attacks requires instrumenting routers with Trusted Platform Modules running signed executables to generate signed data. Attackers have physical access to router hardware and the software source code, so we rely on social measures: we try to deploy to trusted users and assume that they won't collude. Anyone can install BISmark on their own hardware, so we treat measurements from such routers with greater suspicion.

6 Conclusion

Although we did not initially plan to build (and maintain) such a large testbed, we realized the need for it 2009 when we began a study of access network performance. We recognized the variety of uses for a programmable testbed in home networks, and we also discovered that other researchers and operators share our interest. As BISmark continues to expand in terms of size and the diversity of experiments that it hosts, we will need to continually re-evaluate many of our design decisions. We believe our experiences thus far offer a unique perspective in comparison to existing long-running testbeds and useful lessons for others who perform research in broadband access networks.

Acknowledgments

BISmark would not have been possible without its countless contributors. Stephen Woodrow was instrumental in improving the underlying management infrastructure, integrating active measurements with MLab, and managing releases and project outreach. Alfred Roberts and Abhishek Jain led development of Network Dashboard. Thomas Copeland, Adam Allred, Aman Jain, Craig Balfour, and Brian Poole provided excellent backend support. Dave Täht helped bootstrap OpenWrt development. Hyojoon Kim, Abhinav Narain, Sarthak Grover, Andrew Kahn, and Andrew Kim contributed to the codebase. Marshini Chetty, Saad Qaisar, and the PAWS project have supported local deployments in various countries. We thank Giuseppe Aceto, Errol Arkilic, Jonathan Brier, Marc Brown, Enrico Calandro, kc Claffy, Russ Clark, Chip Elliott, Merrick Furst, Ethan Katz-Bassett, Dave Levin, Yogesh Mundada, Michael O'Reirdan, Antonio Pescapé, Bharath Ravi, Tiziana Refice, Glenn Ricart, Swati Roy, Stephen Soltesz, Stephen Stuart, Renata Teixeira, Andy Warner, Meredith Whittaker, and Yiannis Yiakoumis for valuable feedback. This research has been supported by NSF Award CNS-1059350, a Google Focused Research Award, and logistical support from Measurement Lab.

References

[1] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[2] A. Bavier, M. Bowman, D. Culler, B. Chun, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[3] G. Bernardi and M. K. Marina. BSense: a system for enabling automated broadband census. In *Proceedings of the 4th ACM Workshop on Networked Systems for Developing Regions*, 2010.

[4] BISmark Project Partners with Comcast. `http://noise-lab.net/2013/05/19/bismark-project-partners`.

[5] BISmark privacy statement. `http://projectbismark.net/participant/privacy`.

[6] BISmark uploads. `http://uploads.projectbismark.net`.

[7] M. Calder, X. Fan, Z. Hu, E. Katz-Basset, J. Heidemann, and R. Govindan. Mapping the expansion of Google's serving infrastructure. In *ACM SIGCOMM IMC*, IMC '13, 2013.

[8] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: a platform for educational cloud computing. In *ACM SIGCSE Bulletin*, volume 41, pages 111–115. ACM, 2009.

[9] R. Carlson. Network Diagnostic Tool. `http://e2epi.internet2.edu/ndt/`.

[10] M. Chetty, S. Sundaresan, S. Muckaden, N. Feamster, and E. Calandro. Measuring broadband performance in south africa. In *Proceedings of the 4th ACM Annual Symposium on Computing for Development*, DEV 4, 2013.

[11] K. Cho, K. Fukuda, H. Esaki, and A. Kato. The impact and implications of the growth in residential user-to-user traffic. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 207–218. ACM, 2006.

[12] D. Croce, T. En-Najjary, G. Urvoy-Keller, and E. Biersack. Capacity Estimation of ADSL links. In *Proc. CoNEXT*, Dec. 2008.

[13] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proc. ACM SIGCOMM IMC*, San Diego, CA, Oct. 2007.

[14] A. Filastò and J. Appelbaum. Ooni: Open observatory of network interference. In *USENIX FOCI*, Aug. 2012.

[15] J. Gettys. Bufferbloat. `http://www.bufferbloat.net/`.

[16] Glasnost: Bringing Transparency to the Internet. `http://broadband.mpi-sws.mpg.de/transparency`.

[17] Grenouille. `http://www.grenouille.com/`.

[18] S. Grover, S. Sundaresan, M. S. Park, S. Burnett, H. Kim, B. Ravi, and N. Feamster. Peeking behind the nat: An empirical study of home networks. In *Proceedings of the 13nd ACM SIGCOMM conference on Internet measurement*, IMC '13, 2013.

[19] A. Gupta, M. Calder, N. Feamster, M. Chetty, E. Calandro, and E. Katz-Bassett. Peering at the Internet's frontier: A first look at ISP interconnectivity in Africa. In *Passive and Active Measurement Conference*, 2014.

[20] Internet Usage for all the Americas. `http://www.internetworldstats.com/stats2.htm`.

[21] Internet Usage in Asia. `http://www.internetworldstats.com/stats3.htm`.

[22] ITU. Ict facts and figures, Jan. 2012. `http://www.itu.int/ITU-D/ict/facts/2011/material/ICTFactsFigures2011.pdf`.

[23] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[24] G. Maier, A. Feldmann, V. Paxson, and M. Allman. On dominant characteristics of residential broadband internet traffic. In *Proc. Internet Measurement Conference*, Chicago, Illinois, Oct. 2009.

[25] M. Mathis, J. Heffner, and R. Reddy. Network Path and Application Diagnosis. `http://www.psc.edu/networking/projects/pathdiag/`.

[26] Measurement Lab. `http://measurementlab.net`, Jan. 2009.

[27] Network Dashboard. `http://networkdashboard.org/`.

[28] B. Norton. Peering in africa, Aug. 2012. `http://drpeering.net/AskDrPeering/blog/articles/Ask_DrPeering/Entries/2012/8/29_Peering_in_Africa.html`.

[29] OpenWrt. `https://openwrt.org`, Sept. 2013.

[30] Public access wifi service. `http://publicaccesswifi.org/`. Retrieved: September 2013.

[31] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building PlanetLab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 351–366. USENIX Association, 2006.

[32] Project BISmark: Open development portal. `http://projectbismark.github.io`.

[33] RIPE Atlas. `https://atlas.ripe.net`.

[34] S. Roy and N. Feamster. Characterizing correlated latency anomalies in broadband access networks. In *Proceedings of ACM SIGCOMM*, pages 525–526. ACM, 2013.

[35] SamKnows. `http://samknows.com/`.

[36] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: Pushing experiments to the internets edge. In *Proc. of USENIX NSDI*, 2013.

[37] M. Siekkinen, D. Collange, G. Urvoy-Keller, and E. Biersack. Performance limitations of ADSL users: A case study. In *Passive and Active Measurement Conference (PAM)*, 2007.

[38] C. R. Simpson, Jr. and G. F. Riley. NETI@home: A distributed approach to collecting end-to-end network performance measurements. In *Passive & Active Measurement (PAM)*, Apr. 2004.

[39] N. T. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public internet measurement facility. In *Proc. 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2003.

[40] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband internet performance: A view from the gateway. In *Proc. ACM SIGCOMM*, Toronto, Ontario, Canada, Aug. 2011.

[41] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proc. ACM SIGCOMM IMC*, 2013.

[42] uCap: Your data usage assistant. `http://ucap.projectbismark.net/promo/`.

[43] Web attack knows where you live. `http://www.bbc.co.uk/news/technology-10850875`, Aug. 2010.

Application-Defined Decentralized Access Control

Yuanzhong Xu Alan M. Dunn Owen S. Hofmann [*] Michael Z. Lee
Syed Akbar Mehdi Emmett Witchel
The University of Texas at Austin Google, Inc.[*]

Abstract

DCAC is a practical OS-level access control system that supports application-defined principals. It allows normal users to perform administrative operations within their privilege, enabling isolation and privilege separation for applications. It does not require centralized policy specification or management, giving applications freedom to manage their principals while the policies are still enforced by the OS. DCAC uses hierarchically-named attributes as a generic framework for user-defined policies such as groups defined by normal users. For both local and networked file systems, its execution time overhead is between 0%–9% on file system microbenchmarks, and under 1% on applications.

This paper shows the design and implementation of DCAC, as well as several real-world use cases, including sandboxing applications, enforcing server applications' security policies, supporting NFS, and authenticating user-defined sub-principals in SSH, all with minimal code changes.

1. Introduction

Continued high-profile computer security failures and data breaches demonstrate that computer security for applications is abysmal. While there is extensive research into novel security and access control models little of this work has an impact on practice. Instead of applications consistently reimplementing security vulnerabilities, they need a practical and expressive way to use thoroughly debugged system-level primitives to achieve best security practices.

DCAC (DeCentralized Access Control) is our attempt to make modern security mechanisms practical for access control. It has three distinguishing characteristics: it is decentralized in privilege, decentralized in policy specification, and allows application-defined principals and synchronization requirements. Although DCAC greatly increases the flexibility of access control, it retains a familiar model of operation, with per-process metadata checked against per-object ACLs to determine the allowed access. It relies on the standard OS infrastructure of a hierarchical file namespace, extended file attributes, and file descriptors. It is practical for distributed environments because it avoids requiring centralized storage, consistency, or management.

Decentralized privilege. In Linux and Windows, users and groups are principals, and can be assigned privileges. A user might consider creating another user (a "sub-principal") and assigning it a subset of her privileges. This allows an application to run as the sub-principal, and thus with restricted privileges compared to the case where the user directly runs the application. However, on Linux and Windows, administrative functions on users and groups require root privilege. As a result, current OS-level access control does not allow many applications to run with least privilege.

DCAC decentralizes administrator privilege: a normal user can perform administrative operations within her privilege, like creating principals with subsets of her privilege. Privilege separation makes complex applications more difficult to exploit. But current systems require administrative involvement to install and deploy privilege-separated software.

For example, the suEXEC feature of Apache HTTP Server allows it to run CGI and SSI programs under UIDs different from the UID of the calling web server, by using `setuid` binaries. However, creating UIDs for CGI/SSI programs and setting up the `setuid` binaries requires administrator privilege. Not only can use of administrative privilege require human involvement, it also adds opportunities for configuration mistakes that can actually *harm* security. The suEXEC documentation[1] warns the user, *"if suEXEC is improperly configured, it can cause any number of problems and possibly create new holes in your computer's security. If you aren't familiar with managing setuid root programs and the security issues they present, we highly recommend that you not consider using suEXEC."* By contrast, DCAC allows forms of privilege separation, like delegating user privileges to sub-principals, that even in the case of a configuration mistake, limit the effect of a compromise to the privileges of the original user.

Decentralized policy specification. OS-level access control typically defines its principals and policies in a centralized, secure location, such as the `/etc/group` file, the `policy.conf` file in SELinux, or a central policy server (e.g., a Lightweight Directory Access Protocol (LDAP) server). DCAC decentralizes policy specification: policies are stored in files and file metadata at arbitrary locations. DCAC generalizes the `setuid` mechanism of Unix, allowing processes to use the file system

[*] Work completed while at the University of Texas at Austin.

[1] `http://httpd.apache.org/docs/2.4/suexec.html`

to gain fine-grained, user-defined privileges (i.e., not just root). With DCAC, applications control their privileges with a mechanism implemented and enforced by the operating system, but without central coordination.

DCAC is particularly practical for distributed environments, e.g., where machines share a file system via NFS. In such an environment, applications simply use the file system to express access control policy, and any host that mounts the file system will enforce identical access control rules. DCAC does not add its own synchronization requirements, such as entries in a central database. Applications make all access control decisions with access only to their own files. In contrast, a centralized policy server might become a bottleneck when policy queries and updates are frequent, as in many server applications.

Application-defined principals: attributes. Attributes make applications simpler and more secure by allowing them to use access control implemented by the operating system rather than reimplementing their own. Traditional OS principals, such as users, are heavy-weight abstractions that cannot be directly used by applications e.g., a web application that manages its own users.

DCAC *attributes* are hierarchically named strings. Strings are separated into components by the "." character. The string $.u.alice$ can represent the user Alice, but applications are free to define their own encodings and even their own principals. For example, the string $.p.$387.1357771171 might be a principal referring to a process with identifier 387 started about 1.4 billion seconds after January 1, 1970; $.app.browser.password$ might be a component of a browser that is responsible for storing and retrieving the user's passwords. A process may carry multiple attributes simultaneously.

Practicality. DCAC combines ideas from mandatory access control (MAC) systems [7, 16], sandboxing [6], and decentralized information flow control (DIFC) [9, 15, 17, 21, 29] into a practical access control system that is fully backward compatible with current Linux. While MAC and DIFC systems can provide stronger guarantees than DCAC, they require far more effort to use and often struggle with backward compatibility. We believe that application developers will incrementally improve the security of their applications if presented with a simple security programming model that introduces a minimum of new concepts and that can implement security idioms common in modern web-connected applications.

The next section (§2) provides motivating scenarios for DCAC, followed by an extended discussion of design (§3) and relationship to Linux DAC (§4). We describe our DCAC prototype (§5), with a discussion of several applications we modified to use DCAC (§6). We evaluate DCAC (§7), discuss related work (§8) and conclude (§9).

2. Modern access control idioms

We discuss three access control idioms common in today's web-connected applications that are difficult to achieve in modern systems: sandboxing, ad hoc sharing, and managing users. Because these idioms are not well served by current system security abstractions, applications constantly reimplement these idioms, and implement them poorly. Section 3.2 shows how DCAC supports them more naturally than current systems.

Privilege separation/Sandboxing. Suppose Alice wishes to run a photo management program. By default, her program will run with the same privileges as her user account. However, routines for interpreting file formats are often subject to exploitable bugs (such as in the zlib library used to decompress `.png` files [1]). If Alice receives photos from untrusted sources, or even if Alice makes a mistake, all of her potentially sensitive files are endangered, rather than just those that should be managed by her photo management application. Running untrusted or partially-trusted applications has become commonplace, as users frequently download applications from less than trusted sources, or run applications that are often exploitable, such as pdf viewers.

In order to separate her application into a separate privilege domain, Alice must contact an administrator to create a new user (e.g. *alice-photos*), potentially create a new group containing both her and *alice-photos* so that she may easily share files, and install support for running her desired application as the new user, such as via a setuid binary. While it is possible to enforce privilege restriction without superuser support, solutions that do so require complex application-level support (such as in the Chromium web browser [6]) that are easy to get wrong, with disastrous results (e.g., the frequent exploits enabled by bugs in the Java VM's sandboxing mechanism [2, 3]).

This scenario is an example of privilege separation, a well-known technique for building secure applications [20], where each component has only the minimal set of privileges necessary to operate. Privilege separation with OS support is coarse-grained, such as user-controlled namespaces [13] (e.g., namespaces can control access to mount points, but not files or directories).

Ad hoc sharing. Existing systems provide limited facilities for sharing between users. Suppose Alice wishes to share a file or directory with Bob. She can directly send the relevant data to Bob, but if both users wish to be able to update the data they must manually communicate each change. She might change file or directory permissions to allow read/write access to a group that contains both Alice and Bob. Unless this specific group already exists, Alice must rely on a system administrator to create a new group containing both users. If she wishes to add or remove users, she must also rely on a superuser.

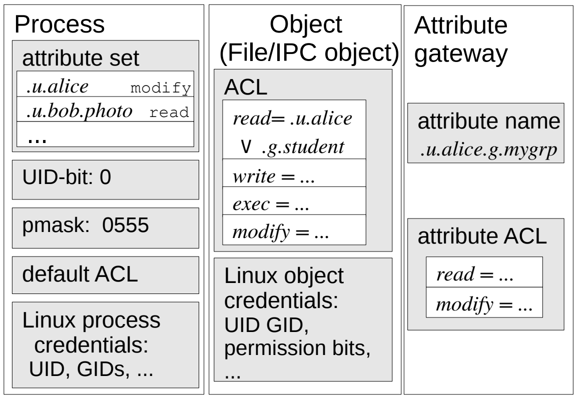


Figure 1: Overview of processes, objects and attribute gateways in DCAC.

If the need to share files expires, an administrator must clean up stale groups.

Facilities such as POSIX ACLs [4] allow users to create more expressive access control lists, by specifying multiple users and groups who may read and write a file. If Alice wishes to share with many users, however, she must either add each user to an ACL for each file to be shared, or again use a group whose membership may only be modified by the superuser. Users should be able to define new groups of users without administrator support, and use those groups in access control on their files.

Managing users. Server applications often have access control requirements similar to those of operating systems, with multiple users having different privileges. However, such applications usually implement access control with manual checks in their own code. Custom security code is a common source of security bugs: a single missed access check can expose sensitive data to unprivileged users. However, most non-trivial applications leveraging OS access control for application security must execute as different OS-level principals and hence require superuser privilege. Such an application must reserve user or group identifiers for its use during installation and possibly during maintenance.

3. Design

In DCAC, the OS enforces simple rules about hierarchical strings (called *attributes*) that are stored in OS-managed process and file metadata. Applications define conventions for what attributes mean to them, allowing them to create access control abstractions that are enforced by the OS.

Hierarchical strings are a self-describing mechanism to express decentralized privilege, where any extension of an existing string represents a subset of the parent's privileges. DCAC attributes have components separated by a "." character: $.u.alice$ is a parent attribute of $.u.alice.photo$. The hierarchy allows regular users to manage principals and policies without requiring a system administrator. For example, Alice may define a new principal for running her photo application (e.g., $.u.alice.photo$) because she owns and controls $.u.alice$.

Each process carries an *attribute set*, which is inherited across process control events such as `fork()` and `exec()`. A process also maintains a *default ACL*. Similar to Linux's `umask`, files created by the process have their access control lists set to the process' default ACL.

Each object (such as a file or shared memory segment) has an *access control list* containing rules that allow processes to access the object based on attributes in the processes' attribute sets. An access control list specifies four *access modes*: read, write, execute, and modify. Each mode has an *attribute expression*, and a process may access a file in a given mode if the process' attribute set satisfies the attribute expression for that access mode. For example, the read access mode might specify $.u.alice \vee .g.student$ which provides read access to user Alice and members of group student (see the read access mode of the object in Figure 1).

Read, write and execute access modes represent the `rwx` permission bits in UNIX-like file systems. The modify mode specifies the permission to modify the file's access control list. In UNIX-like systems, the file's owner implicitly has the right to modify a file's access control.

In DCAC, each access mode has an attribute expression in disjunctive normal form (DNF) without negations. For example, a `jpg` file may have the following access modes:

$$\begin{aligned} read &= (.u.alice.photo) \vee (.u.bob.photo) \\ write &= (.u.alice.photo \wedge .u.alice.edit) \\ exec &= \varnothing \\ modify &= (.u.alice) \end{aligned}$$

If a process has attribute set $\{u.bob.photo\}$, it can read this `jpg` file, but cannot write to it or modify its ACL.

Broadly, DCAC defines rules similar to existing discretionary access control (DAC) in Linux and other systems. A process' attribute set specifies the acting principal, similar to a process' UID and GID. A file's access control list specifies which principals may access the file, similar to permission bits, which in conjunction with a file's owner and group specify access rights for a UID and GID. However, DCAC is significantly more flexible and decentralized. A process can have many attributes in its attribute set without differentiating between users and groups, files can specify formulae for allowing processes access, and there is no central mapping between string identifiers understandable by the user and integers understandable by the system.

Note that although we use $.u.\langle user\rangle$ and $.g.\langle group\rangle$ throughout the paper to couch our discussion in terms of users and groups, DCAC does not enforce any attribute naming scheme. Users and groups in DCAC exist only by convention.

3.1 Adding and dropping attributes

DCAC allows a process to change its access control state by modifying its attribute set. Attribute set modification allows users and applications to run different processes with different privileges. A process p can always drop an attribute from its attribute set, but to add an attribute, it must satisfy one of the following rules:

- p has the parent of the requested attribute (e.g. $.u.alice$ is the parent of $.u.alice.photo$).
- p has permission to use an *attribute gateway* (discussed later in this section).

A process can add attributes to its attribute set in one of two modes, *read* or *modify*, depending on its attribute set and the configuration of attribute gateways. Read mode enables a process to use an attribute. Modify mode adds control over granting the attribute to other processes. A process with an attribute in modify mode can always downgrade that attribute to read mode. A process cannot upgrade an attribute to modify mode without a gateway or a parent attribute in modify mode. When extending an attribute by adding a new component, the extended attribute has the same mode as the parent.

Allowing a process to create and add attributes based on hierarchy permits regular users to create new principals without requiring administrator privileges. To enable flexible, application-defined resource sharing, processes use attribute gateways to acquire attributes based on user-defined rules.

Decentralized policy via attribute gateways. A primary design goal for DCAC is decentralization. Instead of centralized credentials (e.g., `/etc/group`) or centralized access policy (e.g., SELinux's `policy.conf`), DCAC distributes credentials and access policy using *attribute gateways*, which are a new type of file (in our Linux implementation, they are empty regular files with specific extended attributes). An attribute gateway allows a process to add new attributes to its attribute set based on its current attribute set. An attribute gateway is a rights amplification mechanism, like a `setuid` binary, but more flexible.

An attribute gateway has only two access modes, read and modify (execute and write are not used). If a process' attributes fulfill an access mode for a gateway, then the process may add the attribute to its set in that mode. For example, Alice might tell her colleagues that she is starting a group with attribute $.u.alice.g.atc$ for documents related to a submission to USENIX ATC. She sends email to the members of the group explaining that there is a gateway for the group in a file called `~alice/groups/atc.gate`. Her collaborators modify their login scripts to open that attribute gateway as part of their login process. The gateway's read access mode consists of a disjunction listing the user attributes for the group's members (e.g., $read = (.u.alice) \vee (.u.bob) \ldots$).

Gateways decentralize credentials and access policy. Multiple gateways (or no gateways) may exist for any attribute, with the location of the gateways and their access controlled by users and convention. DCAC does not force use of a central repository of credentials or access policy, though users may choose to create and use centralized repositories.

Having an attribute in modify mode allows a process to decide which other principals can obtain the attribute, just as having modify access to a file allows a process to control which principals can access the file. Specifically, having an attribute in modify mode allows a process to change the access control list for any gateway for the attribute, or to create new gateways for the attribute.

Access policies should incur performance penalties only for features they actually use [26]. With a hierarchical attribute namespace and decentralized attribute gateways, different users and applications can perform administration in their own domains without contention; in contrast, any administrative operations on UIDs and `/etc/passwd` (for example) require central coordination across the system.

Gateway management. Gateway management is up to users and their applications. Since DCAC is a generic kernel-level mechanism, it does not impose requirements for gateway locations; however, specific system deployments or applications can follow conventions such as a central repository of gateways. (Similarly, the Linux kernel does not enforce the use of the `/etc/passwd` file, making it general enough to support Android's application-based access control.) While users can harm themselves with incorrectly set gateway permissions, gateways do not add problems beyond those of current systems as there are already opportunities for self-harm with standard file permissions. For example, a user can make his or her `ssh` private key file world readable.

DCAC allows gateways to be in any location, which could result in a less regulated environment than in centralized systems. With current file permissions it is relatively easy (modulo perhaps setuid binaries) to determine exactly which users and groups may (transitively) access a file. Attribute gateways require an exhaustive search of the filesystem to find all attributes that might allow a given process access to a file. On the other hand, because it is easier to run processes with reduced privilege, a user could restrict her programs to only create gateways in specific, relevant directories.

Summary. We believe DCAC achieves a new balance of expressivity and simplicity due to the features:

- The attribute abstraction is generic. An attribute can represent a user, a group, a capability, an application or a category of files, depending on the user or the application's need.

- Attributes are hierarchically named, making privilege delegation possible by extending existing attributes.
- Decentralized attribute gateways allow processes to acquire attributes that they would otherwise not be able to acquire strictly from attribute hierarchy. Creation of and policy for gateways is controlled by users and applications.
- Attributes are self-explaining strings. There is no need to map attributes to other OS-level identifiers like UIDs. Identical strings from different machines refer to the same attribute, making attributes directly sharable. This enables DCAC to support NFS (§6.3) with minimal development effort.
- There is no rigid distinction between "trusted" and "untrusted" processes. Instead, process access boundaries can be flexibly defined and flexibly delegated.

3.2 DCAC supports modern access control idioms

Here, we describe how DCAC supports the modern access control idioms described in §2.

Privilege separation/Sandboxing. Suppose Alice wishes to run her photo manager in a separate, restricted environment. Alice invokes her photo manager with a simple wrapper that does the following:

1. Adds a $.u.alice.photo$ attribute (allowed because Alice's process runs with the $.u.alice$ attribute).
2. Drops the $.u.alice$ attribute.
3. Executes her photo application.

She may similarly run her PDF reader in a separate, restricted environment by following the same steps with a $.u.alice.pdf$ attribute. Alice then sets up ACLs to allow processes running with $.u.alice.photo$ to access her photo manager's files, and $.u.alice.pdf$ to access her pdf reader's files. Each application may now access only its own set of files. In §6.2, we show how DCAC helps to sandbox an application (Evince) with vulnerabilities.

A DCAC-aware application may also enable finer privilege separation between different components. Suppose Alice's photo application wishes to isolate its file decoding routines. It may run those routines in a separate process which carries the $.u.alice.photo.reader$ attribute, and drops the $.u.alice.photo$ attribute. The program can grant $.u.alice.photo.reader$ read-only access to the photo files but nothing else, to prevent an exploit from reading other files or writing any file.

Ad hoc sharing. Hierarchical attributes combined with policies for adding attributes to a process' attribute set allow regular users to customize how they share files. Suppose that Alice (whose processes carry the $.u.alice$ attribute) wishes to share a file or set of files with a group of users including Bob (see Figure 2). Rather than updating each file every time she wants to share with a new user, like Bob, Alice instead does the following:

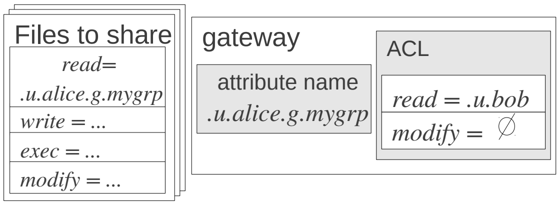


Figure 2: Ad hoc sharing with DCAC. Alice shares files by allowing $.u.alice.g.mygrp$ read access. She then creates an attribute gateway allowing $.u.bob$ to add the attribute.

1. She creates a new attribute $.u.alice.g.mygrp$.
2. Alice creates a new attribute gateway for $.u.alice.g.mygrp$ (e.g., `~alice/groups/mygrp.gate`) and sets its attribute formula for the read access mode to $(.u.bob)$. Bob must learn the location of this file.
3. Processes running as Bob can use Alice's new group by locating the attribute gateway, and using it to add the attribute $.u.alice.g.mygrp$ to their attribute set. These processes will be allowed the relevant access to any file with an ACL that matches $.u.alice.g.mygrp$.

Alice can change membership of her group via ACLs on the attribute gateway. Note that the ad hoc group is a set of attributes; besides OS users, they may also represent any application-defined principals. Ad hoc sharing is used in our modified DokuWiki server in §6.1.

Managing users. Consider a server application that runs processes for multiple users, and stores user data in files. The server can use DCAC to implement access control by assigning different attributes to different users' processes, allowing application-level security requirements to be easily expressed as string attributes. For example, Linux user Alice runs `myserver`. A user on the server, `webuser`, is assigned an attribute set that has only $.u.alice.myserver.u.webuser$. A server process serving `webuser`'s requests can only access files which allow access to $.u.alice.myserver.u.webuser$. If the processes need to share some common files, the server can add a common attribute $.u.alice.myserver.common$ to their attribute sets, and add the attribute to the files' ACLs. Our modified DokuWiki server (§6.1) manages user and group permissions this way.

If a server application executes on multiple machines, it only needs to synchronize on relevant policies under its management, instead of updating them for the whole system. Server applications do not need to synchronize on a map between application-level users and OS-level access control state.

We apply DCAC to NFS (see §6.3). While different machines still share a set of OS users and groups that are already under centralized management, application-defined principals and policies do not require centralization.

Sub-principals delegating to sub-principals. DCAC allows flexible delegation to sub-principles, addressing one of the most vexing problems created by many application-specific user management systems (such as `sshd` and `apache`).

Consider the following: Professor X and Professor Y wish to collaborate. Y sends a credential (like a public key) to X and X uses it to add Y as a sub-principal. Y can now access resources shared by X, such as a subversion repository holding a joint publication. However, Professor Y recruits graduate student Z to actually do the work. Most sub-principal systems are not flexible enough to allow Y to delegate to Z without giving Z his credential. Therefore, Y must talk to X and make him aware of Z. X's list is a centrally administered bottleneck.

DCAC allows principals in a service to define and manage their sub-principals, without registering them with the service in a centralized way. Using DCAC, Professor Y can provide Z a login without involving Professor X and without revealing his credential to Z.

We let a local user, X, have his own program, `sub-auth`, to authenticate his sub-users. The program is located at a known per-user location in the system, such as `~X/sub-users/sub-auth`. X's `sub-auth` program can define where the credentials of his sub-users are stored, and how to authenticate sub-sub-users. For example, it may use a file to store sub-users' credentials, and delegate authentication to the sub-user. Thus, Y, a sub-user of X, would have his own `sub-auth` program to authenticate Y's sub-users. §6.4 shows a detailed example of delegation for sub-users using `sshd`.

DCAC does not require `sub-auth` programs, nor does it require particular naming conventions for them. A user may authenticate sub-users with a chain of certificates provided during login. A sub-user would require a certificate chain of length one: a local user vouching for the sub-user. A sub-sub-user must provide a chain of length two, and so on. Such a scheme does not require local storage for credentials and each user must only vouch for (and know about) their direct sub-users.

4. Harmonious coexistence of DCAC and DAC

DCAC is designed to work harmoniously with Linux's existing discretionary access control system (DAC), but it has mandatory access control (MAC) mechanisms to express policies more nuanced than what can be expressed by DAC alone. DCAC can augment a process' rights (e.g., allow Alice's calendar process to read a file Bob's process wrote and shared with Alice), but it can also restrict rights (e.g., limit Alice's photo reader from reading her email).

Augmenting Linux DAC. A DCAC system is permissive by default, allowing access if DAC or DCAC checks succeed. By default, files have empty ACLs, so access checks reduce to DAC checks. As a result, all valid Linux disk images are valid DCAC disk images. By preserving DAC permissions, DCAC can be deployed incrementally.

Allowing access if DAC or DCAC checks succeed makes sharing easy. For example, Alice may share a file with Bob by simply adding $.u.bob$ to the file's ACL. If instead we required DAC *and* DCAC access checks to succeed, Alice would have to adjust permissions on many of her files to start using DCAC (e.g., she would have to make her `authorized_keys` readable by sshd).

Restricting Linux DAC. Using DCAC for restricting Linux DAC permissions requires two additional pieces of state, a **pmask** (permissions mask), and a **UID-bit**. Each process has a `pmask` and a `UID-bit` that are inherited by child processes after a `fork()`, and are maintained across `exec()`. A process can only change its `pmask` by making it more restrictive (i.e., by clearing bits), and it can only clear the `UID-bit`.

The permissions mask is intended to prevent a process from reading or writing resources that Linux's DAC permissions allow (e.g., because the UID of a process and file owner match, and the file owner has read access). The permissions mask is ANDed with standard DAC permissions bits before each permissions check. So a DCAC system will check (`perms` & `pmask`) instead of `perms`. Therefore, `pmask` = 0777 does not restrict Linux DAC, while `pmask` = 0755 restricts write access to other users' files, and `pmask` = 0 causes all DAC permission checks to fail.

Each process has a `UID-bit` which is intended to limit the ambient authority granted to a process when its UID matches the UID of a resource. For example, currently in Linux, a process may change the permissions on a file or directory with matching UID even if it has no permissions on the file or its containing directory. If the `UID-bit` is clear, the process is restricted in the following ways:

1. It can only change the DAC permissions and DCAC ACLs of files with matching UID if it satisfies the modify access mode.
2. `kill` and `ptrace` are restricted to child processes.
3. It cannot remove an IPC object, change permissions on it, or lock/unlock a shared memory segment, unless the DCAC check for the object's modify access mode succeeds.

A process with the `UID-bit` set may modify DAC permissions and ACLs for files and directories with matching UIDs. An unrestricted process (e.g., running as root) with the `UID-bit` set can change DAC permissions and DCAC ACLs on any file or directory. It can also can send signals to any process.

5. Prototype implementation

We implement a DCAC prototype by modifying Linux 3.5.4.

5.1 Programming interface

In DCAC, attributes are managed through file descriptors. When a process successfully adds an attribute, it receives a file descriptor for that attribute. The corresponding kernel file object is a wrapper for the attribute. We use system calls `open`, `openat` and `close` to add and drop attributes.

- `int openat(int fd, char *suffix,`
  `int flags)`:
  If `fd` represents an attribute *attr* in the attribute set, this call adds *attr*.⟨**suffix**⟩ and returns the associated file descriptor. `flags` can be `O_RDONLY`, or `O_RDMOD`, representing that the requested access mode is read or modify (which subsumes read). If the parent attribute has only read mode, requests for modify access mode on the new attribute will be denied.
- `int open(char *pathname, flags)`:
  If the file at `pathname` is an attribute gateway, DCAC will evaluate the access modes according to `flags` (`O_RDONLY` or `O_RDMOD`). On success, the attribute is added and the corresponding file descriptor is returned. Note that this operation does not open the actual gateway file.
- `int close(int fd)`:
  If `fd` represents an attribute, that attribute is dropped.

A potential issue with using file descriptors is compatibility with applications which are not aware of DCAC. Sometimes applications close all open file descriptors as a clean-up step (since file descriptors persist on fork and exec), which results in unintended dropping of attributes. To address this problem, we add a `lock` flag to each process' DCAC state. When the flag is set, the process' DCAC state cannot be changed, until the flag is cleared. One can set this flag in a wrapper and then invoke the application (see §6.2). The lock flag is intended solely to ease backward compatibility.

A related complication of using file descriptors to represent attributes is that programs may set the `close_on_exec` flag on their open file descriptors. For instance, if `bash` is started as a login shell (where it needs to load the user's custom settings), it sets the `close_on_exec` flag on all file descriptors except the standard I/O streams. This can causes attributes to be dropped unintentionally. DCAC ignores `close_on_exec` for attributes.

Besides `open`, `openat`, and `close`, all other operations are encoded into 4 new system calls. Table 1 shows the DCAC API.

We additionally wrote a 274-line[2] SWIG[3] wrapper to make DCAC functionality available in PHP.

5.2 Processes and Objects

The core functionality of DCAC is implemented as a Linux security module (LSM [28]).

Processes. LSMs use a `security` field in the Linux per kernel thread `task_struct` structure to store the security context of a process. The DCAC state for a kernel thread includes the attribute set, default ACL, `pmask` and the `UID-bit`, all of which are stored in a structure pointed to by the `security` field (see §4).

In a multi-threaded application, each thread can have its own DCAC state. Threads can thus run on behalf of different principals, and access control decisions are based on their individual DCAC state, which would be useful for a trusted, uncompromised server application. However, running untrusted code in a thread can lead to loss of isolation between principals in case of a compromise as threads share the same address space.

Files. For persistent file systems, DCAC requires support for extended attributes. File permission (read, write, execute, and modify) ACLs are stored in files' extended attributes, with the entire ACL encoded in a single extended attribute. For attribute gateways, the attribute controlled by the gateway and its ACLs (read and modify) are encoded in a single extended attribute.

Permission checks happen only when files are opened and not on subsequent reads/writes. The permission check occurs in the `inode_permission` LSM hook. The `inode_permission` hook is a *restrictive* hook, which means it cannot grant access if a request is already denied by the Linux DAC. However, DCAC allows access if either DAC or DCAC is satisfied (§4). Therefore, in addition to LSM, we modify 4 lines of code in `fs/namei.c` to achieve DCAC's semantics.

ACL cache. DCAC keeps a generic, in-memory ACL cache for each file in the VFS layer. There are two motivations for such a cache. First, it reduces performance overhead for remote file systems (e.g. NFS). Second, it makes DCAC usable for non-persistent file systems (e.g. sysfs). The in-memory `inode` structure contains an `i_security` field, where DCAC stores the ACL cache. The cache is initialized (from the file's extended attributes) when the ACL is first needed: when a DAC permission check fails. The cache also records each file's change time (ctime) when the ACL is fetched. The cache provides a mechanism for file systems to invalidate ACL entries to enforce filesystem coherence semantics. The ctime value in each cache entry can be used to deter-

[2] All line counts: `http://www.dwheeler.com/sloccount/`

[3] `http://www.swig.org`

Sys call	API	Functionality

`dcac_add`	`int dcac_add_any_attr`
	`(const char *attr, int flags)`
		Add the the attribute `attr`, for root user only.

`dcac_acl`	`int dcac_set_def_acl`
	`(const char *dnf, int mode)`
		Set an access mode, specified by `mode`, in the process' default ACL to `dnf`.

	`int dcac_set_file_acl(const char *file,`
	`const char *dnf, int mode)`
		Set one access mode in the file's ACL to `dnf`.

	`int dcac_set_attr_acl(int afd, int ffd,`
	`const char *read, const char *mod)`
		Create/change a gateway. `afd` and `ffd` are file descriptors of the attribute and the gateway file.

`dcac_info`	`int dcac_get_attr_fd(const char *attr)`	Get the file descriptor of the attribute `attr`.

	`int dcac_get_attr_name`
	`(int fd, char *buf, int bufsize)`
		Get the string representation of the attribute associated with `fd`.

	`int dcac_get_attr_list`
	`(int *buf, int bufsize)`
		Store the file descriptors of all the attributes of the process to `buf`.

`dcac_mask`	`int dcac_set_pmask(short mask)`	Set `pmask` to (`pmask` & `mask`).
	`void dcac_clear_uid_enable(void)`	Clear the `UID-bit.`
	`void dcac_lock(void)`	Lock the process' DCAC states.
	`void dcac_unlock(void)`	Unlock the process' DCAC states.

Table 1: DCAC API.

mine whether the entry needs to be invalidated, because changing the extended attribute causes a ctime change.

IPC objects. The Linux kernel's IPC object data structures share a common credential structure, `kern_ipc_perm` [28], where the DCAC ACL is stored. DCAC checks permissions to access these objects in LSM multiple hooks. We changed 6 lines in `ipc/utils.c` to allow access if DAC or DCAC allows it.

5.3 Attribute management

Using the rules described in §3.1, a process can only add new attributes based on existing attributes in its attribute set. To initialize attribute state, our Linux DCAC implementation allows processes running as root to add any attribute to their attribute sets with arbitrary modes (read or modify). We then modify system binaries, such as `login`, to initialize the attribute state for user processes. `login` is already responsible for initializing a process' UID and GID state by reading the `/etc/passwd` and `/etc/group` files and invoking system calls such as `setuid` and `setgroups`. We extend this responsibility to include attributes.

Our examples use $.u.alice$ to represent an attribute corresponding to a specific Linux user. We encode this convention in our prototype by changing `login` to add the $.u.\langle username\rangle$ attribute, with both read and modify access modes, to user login shells. Similarly, $.g.\langle grpname\rangle$ attributes represent Linux groups, and they are added with only read mode, since only root has administrative control of them. Modifying `login` required a 28 line change to the shadow 4.1.5.1 package.

We also modify `sshd` and the LightDM desktop manager[4] to set up the attribute state when the user logs in remotely or via a graphical user interface. We changed 37 lines of code in OpenSSH 5.9 and 20 lines of code in LightDM 1.2.1.

6. DCAC application implementation

We demonstrate several use cases of DCAC in real applications.

6.1 Application-defined permissions in DokuWiki

DokuWiki[5] is a wiki written in PHP that stores individual pages as separate files in the filesystem. As a result, OS file-level permissions suffice for wiki access control. In fact, access control only requires setting attributes and default file ACLs on login. Then the OS ensures that all ACL checks occur properly, without need for application-level logic.

We add a 246-line DCAC module to DokuWiki's collection of authentication modules. DokuWiki with DCAC executes in a webserver initialized with the $.apps.dokuwiki$ attribute. Upon user login, the webserver process acquires a user-specific attribute $.apps.dokuwiki.u.\langle username\rangle$ and drops $.apps.dokuwiki$. Default ACLs on all created files are set to the user-specific attribute. The server permits anonymous page creation and access through a common attribute $.apps.dokuwiki.common$. DokuWiki runs as a CGI script to ensure a new process with the $.apps.dokuwiki$ attribute handles each request, restricting the impact of a compromise during a request to the logged in user. Otherwise, a reused compromised process could affect other users when it acquires their attributes during a new request.

DokuWiki has a built-in ACL system that is only controllable by superusers. We modify DokuWiki to support user-created groups. To do this, we create a

[4] `http://wiki.gentoo.org/wiki/LightDM`

[5] `http://www.dokuwiki.org`

new directory to hold gateway files with a directory per wiki user. When a user creates a group, she also creates a gateway file to attribute $.apps.dokuwiki.u.\langle username\rangle.g.\langle groupname\rangle$ with the name of the group in a per-user location defined by a DokuWiki naming convention. The gateway file has read permission for the members of the group. Each user's group directory is traversable (has execute permission) by all users. When a user is informed (out of band) that she has been added to a new group, she records the group name and gateway path name in her groups file. We have a user modify her own groups file to ensure that her groups are not disclosed by arbitrary access to a common file. During login, the DCAC authentication module reads the user's groups file and attempts to access gateways and add group attributes.

We add calls to DokuWiki's XML RPC interface to use this group functionality: `setPerms(pageName, perms)` allows the DCAC permissions of wiki files to be adjusted. `setGroup(groupName, members)` modifies the membership of the group named `groupName`. Finally, `getMyGroups` and `setMyGroups` allow a user to activate the additional privileges given to her by groups by modifying her groups file.

6.2 Sandboxing Evince

We implement an application wrapper that sets up DCAC state for an application such as the default ACL, `UID-bit`, `pmask` and attribute set. The wrapper can perform user-specified work (via scripts) before application execution (e.g., granting a sandboxed application permissions to a specific file) and after application termination. With a wrapper, DCAC can sandbox unmodified applications because DCAC state is inherited across `fork` and `exec`.

We port a simple stack-based buffer overflow targeting an old version of Evince (evince-0.6.1) to test the sandboxing ability of DCAC. Since document viewers generally do not need write permission, we can sandbox Evince by clearing its `UID-bit` and setting its `pmask` = 0115, by using the application wrapper. We allow execute permission for directory traversal, and allow read for world-readable files so Evince can load its shared libraries. Additionally, to allow evince to open the target `pdf` file, the wrapper keeps $.u.alice.evince$ in the attribute set, and adds it to the `pdf` file's ACL. The wrapper can reset the file's ACL after Evince terminates. Upon triggering the exploit, the attacker only has access to world readable files, and even a shell opened via an exploit remains confined.

6.3 NFS

In a normal NFS environment, machines are within the same trust domain, and share a common set of OS users and groups, which usually requires centralized management. By adopting an attribute naming convention for users and groups (such as $.u.\langle username\rangle$ and $.g.\langle grpname\rangle$), DCAC eliminates centralized management. A user can define her own sub-principals and manage them in her own way, on all machines.

The Linux NFS implementation does not support extended attributes, but we ported a patch for NFSv3 [18] to add extended attribute support. In addition to the patch, we also modified 326 lines of code for NFS in the Linux kernel source.

The NFSv3 specification [24] does not define how a server should check permissions. In the NFS implementation in Linux, a client OS checks permissions when a file is opened, but the server does not check permission for subsequent `read` calls, or for `write` calls on the files that belong to the process' user. It only checks permission for `write` calls on files that belong to a different user. We remove this extra check in DCAC. We believe that this change is sensible, since under this change NFS files still obey the standard UNIX convention where permissions are checked only on open.

Clients must make DCAC access control decisions, as they have access to a process' attributes as well as the resource's (e.g., file's) attributes. However, creating or removing files and directories requires write access to the parent directory and in Linux NFS the client simply forwards these operations to the server without checking permissions. In DCAC, if a client uses attributes to determine that a `create` or `remove` operation is legal, it appends a hash of the cached ACL for the parent directory in its RPC to the server. The server checks the hash against the ACL, and if they match, it knows the client has an up-to-date copy and can trust the client's decision to allow the operation. While NFS servers trust their clients, this check is to ensure that clients do not make wrong decisions based on stale permissions. In addition, DCAC appends the process' default ACL to `create` and `mkdir` calls, to initialize ACLs on newly created files and directories.

For regular files, DCAC also uses the ACL cache to determine permission when they are opened. To guarantee *close-to-open* consistency [24], the cache is invalidated when a ctime change is observed.

6.4 Managing sub-users in SSHD

We modify 81 lines of `sshd` to support the access control model described in Section 3.2.

Modern versions of `sshd` support a *forced command* option, which allows unprivileged users to authenticate sub-principals with public keys via the `svnserve` program. Arguments to `svnserve` control details like the user name for sub-principals (because sub-principals do not have user names in `/etc/passwd`). However, `svnserve` does not allow the kind of flexible delegation described in the next example.

Authentication example. When `sshd` receives a login request for X.Y, it invokes X's `sub-auth` program with only X's privilege, and passes to it the sub-user name "Y" and the credentials the request provides. If the `sub-auth` program returns successfully, `sshd` approves this request and restricts X.Y's privilege by properly setting the attribute set, `pmask` and `UID-bit`.

For a more concrete example, consider a hierarchy of users: X.Y.Z, described here and illustrated in Figure 3.

1. When X.Y.Z tries to login using `ssh`, he provides username "X.Y.Z" and some credential.
2. `sshd` invokes X's `sub-auth` program, passing sub-username "Y.Z" and the credential to it, with UID set to X's and only $.u.X$ in the attribute set.
   (a) X's `sub-auth` finds that it is one-level down, it keeps only $.u.X.Y$ in the attribute set, and restricts `pmask` and `UID-bit`.
   (b) It `exec`'s Y's `sub-auth`, passing sub-username "Z" and the credential to it. Y's `sub-auth` verifies the credential, and returns successfully.
3. Now `sshd` knows the request is authenticated. Before `exec`'ing the shell, `sshd` keeps only $.u.X.Y.Z$ in the attribute set, and restricts `pmask` and `UID-bit`.

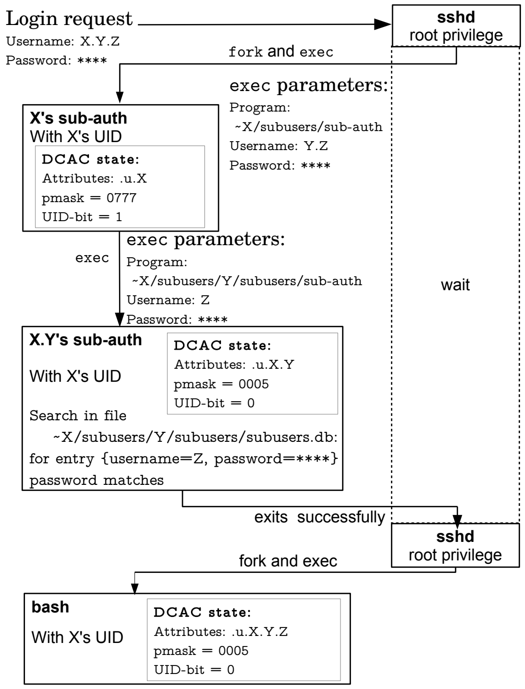


Figure 3: Authentication of sub-users in our modified `sshd`: support for arbitrary nesting of sub-principals.

Note that another OS user, A, can have a completely different `sub-auth` program. His `sub-auth` program may be based on certificate chains, and does not need further lower-layer `sub-auth` programs. For example, A can sign a certificate for B's public key as A.B, and B can sign another certificate for C's public key as A.B.C. When A.B.C logs in, he needs to provide the two certificates to be verified by A's `sub-auth`, as well as a proof that he has C's private key, such as a signature.

7. Evaluation

We measure the performance overhead of DCAC through both targeted benchmark programs and representative applications. Our benchmarking systems had quad-core Intel Core2 2.66 GHz CPUs, 8 GB of RAM, and a 160 GB, 7200 RPM disk. All servers and clients were connected by gigabit Ethernet.

7.1 Microbenchmarks

Filesystem. We run the Reimplemented Andrew Benchmark (RAB) [19], a version of the Andrew benchmark scaled for modern systems, on both a local ext4 filesystem and NFSv3.

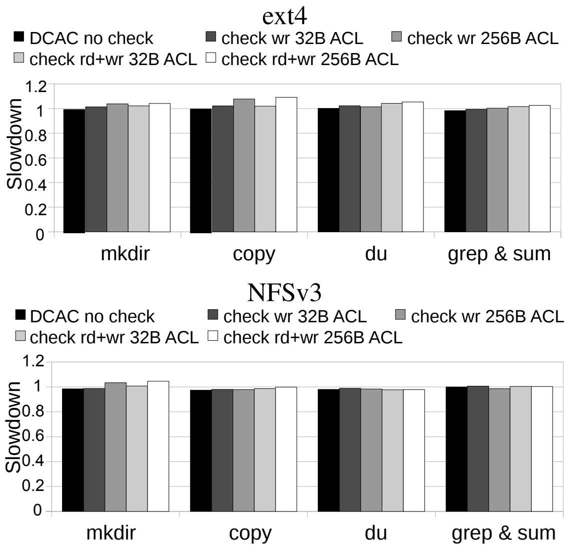


Figure 4: RAB results on local ext4 and NFSv3. 20,000 directories are created in the mkdir phase, and 100 files of 1 KB each are copied to 500 directories in the copy phase. The slowdown is relative to unmodified Linux.

RAB initially creates 100 files of 1 KB each and measures the time for the following operations: (1) creation of a number of directories, (2) copying each of the 100 initial files to some of these directories, (3) executing the `du` command to calculate disk usage of the files and directories, and (4) using `grep` to search all file copies for a short string and checksumming all the files. The exact number of operations varied depending on category (ext4 or NFS) and is described with the corresponding figures. Results are shown in Figure 4.

We compare the results for the following cases:

- The baseline, which uses an unmodified Linux kernel.

FS	Time (μs)		
	chmod	Changing DCAC file ACL	
		32B ACL	256B ACL
ext4	1.24	2.19	4.27
NFSv3	224	228	238

Table 2: Time to change the ACL of a file, compared to `chmod`.

Setup	Baseline	DCAC
Time (ns)	279	355 (1.27×)

Table 3: Overhead for a semaphore operation.

FS	Time (s)		
	baseline	check wr	check rd&wr
ext4	197.3	198.3 (1.01×)	201.2 (1.02×)
NFSv3	322.1	325.4 (1.01×)	325.7 (1.01×)

Table 4: Kernel compile time, averaged over 5 trials. "check wr" means the DCAC write access modes on output files and directories are checked; "check rd" means the DCAC read access modes on the source files are checked. "baseline" means using unmodified kernel. The size of each ACL is 256 bytes.

- DCAC kernel, where the default ACL is empty, and ACLs on files are not checked. DCAC adds at most 1% overhead on local ext4 and NFS.
- DCAC kernel, where the default ACL only contains the write access mode; DCAC ACL checks occur for every write access. On local ext4, the overhead is 0% to 4% for a 32B ACL, and 0% to 8% for a 256B ACL. On NFS, the overhead is below 4%.
- DCAC kernel, like the prior case, but DCAC ACL checks occur for both read and write accesses. On local ext4, the overhead is 1% to 4% for a 32B ACL, and 2% to 9% for a 256B ACL. On NFS, the overhead is below 5%.

On ext4, the kernel stores extended attributes within inodes if they are below a certain size threshold. With 256-byte inodes, 32 bytes is below this threshold; 256 bytes is not, so extra disk blocks must be allocated, resulting in larger overhead in the mkdir and copy phases, where new files are created. For the du phase and the grep and sum phase, observed differences correspond to whether read ACLs are being checked and differ little for ACL sizes. This is likely because DCAC reads ACLs from ACL caches for most of the time.

On NFS, the number of round trips per operation dominates the performance; overhead for disk storage for extended attributes on the server is negligible in comparison. Most of the time, DCAC reads ACLs from the ACL caches, which does not incur network communication. On the creation of a file or directory, DCAC appends the initial ACL to the `create` or `mkdir` RPC, instead of using a separate `setxattr` RPC. As a result, DCAC adds very small overhead on NFS.

ACL manipulation. We compare the time it takes to change the ACL of a file, to `chmod` in Linux. Table 2 shows that, on a local ext4, changing ACLs can be 1.7× to 3.4× slower than `chmod`, depending on the size of the ACL; on NFSv3, ACL size has a small impact on performance, and the time spent for changing ACLs is comparable to `chmod` (under 6.5%).

IPC. DAC and DCAC check permission for shared memory segments and named pipes only when they are attached to the process' address space or opened; however, every up or down operation on a System V semaphore requires a permission check. We measure the overhead induced on semaphore operations by DCAC ACL checks. The baseline comes from the DCAC kernel where DAC permission checks always succeed, so no ACL checks ever occur. The DCAC measurement comes from removing a process's DAC permissions by setting its `pmask` to 0 and giving it an attribute. The semaphore is accessible to processes with this attribute. Thus, we ensured that a DCAC ACL check is performed on every semaphore operation. We measure the average time per semaphore operation over a long sequence that alternated between up and down (measuring the average overhead of both), and set the initial semaphore value so that no semaphore opertion blocks. Table 3 shows 27% slowdown for a semaphore operation requiring a DCAC permission check.

7.2 Macrobenchmarks

Kernel compile. We measure the time to compile the Linux kernel (version 3.5.4, without modules). Table 4 shows the overhead is negligible for both an ext4 filesystem and NFS. DCAC only performs additional permission checks on file open, creation, and deletion. The amount of time spent on these operations is small enough compared to the computation involved in a kernel compile to cause low overhead.

DokuWiki. We benchmark DokuWiki by playing back a set of modifications made to the DokuWiki website (which is itself run using DokuWiki). This is a set of 6,430 revisions of 765 pages. We made a set of requests to a wiki with a 90% read workload. Each write operation replaces a page of the wiki with the next version in the set of revisions that we have. We measured the total wall clock time for 16 clients to perform 100 requests apiece against the wiki. The baseline is the wiki running on the same machine with the same kernel but no attributes applied to any files and using standard authentication. Results are in Table 5. DCAC authentication and plain authentication results were within margin of error of each other. This is expected, as DCAC merely adds a few system calls to operations that otherwise have a lot of computation and file I/O through running PHP scripts.

Setup	Baseline	DCAC
Time (s)	45.5 ± 0.7	45.3 ± 0.7

Table 5: Wall clock times for 16 clients to complete 100 requests apiece to DokuWiki. Standard deviations are determined from 10 trials.

Systems	Relation to DCAC
Sandboxes	Focus on isolation, DCAC also accommodates fine-grained sharing
Flexible policy specification	Focus on completeness rather than usability, DCAC strives for balance
Application- and user- defined access control	DCAC provides fine-grained control, supports network filesystems
New security models	DCAC concepts easier to understand: closer to traditional users and groups

Table 6: Comparing DCAC with related systems.

8. Related work

DCAC is most directly inspired by two systems, Capsicum [27] and UserFS [14]. Capsicum shows that programmers want and will use system abstractions that make writing secure code easier. Capsicum implements a fairly standard capability model for security; its innovation is in casting file descriptors, an abstraction familiar to Unix programmers, as a capability. DCAC applies this insight by representing attributes as file descriptors. Capsicum's capability mode is similar to DCAC's `pmask` and DCAC's `UID-bit`, in that they are both used to deprivilege a process and restrain its ambient authority granted by legacy access control systems; however, DCAC's `pmask` and `UID-bit` are more fine-grained – they can selectively restrict a process' ability to perform different operations on different files.

UserFS [14] leverages existing OS protection mechanisms to increase application security by explicitly maintaining a hierarchy of UIDs to represent principals. Unfortunately, system-wide UIDs are awkward for dynamic principals. For example, independent server applications would contend for UIDs even though their principals are in logically separated domains. Moreover, in a distributed setting, groups of machines would need to synchronize on what UIDs are currently in use. DCAC is designed to work well where UserFS struggles—highly dynamic, distributed deployments within a single administrative domain.

There are too many access control systems and proposals to analyze them all, so we describe the novel combination of features in DCAC by contrasting with entire classes of access control systems, with modern exemplars. Table 6 summarizes our analysis.

Sandboxing. Many projects try to isolate ("sandbox") potentially malicious code from the rest of the system. Android repurposes UIDs to isolate mutually distrusting applications from one another. The Mac OS X Seatbelt sandbox system can constrain processes according to user-defined policies, which is used by Chromium [8].

User-level sandboxes are often specific to an application, and are hard to get right because applications regularly change the files and directories they access. As a result they suffer problems with usability and security vulnerabilities [2, 3].

DCAC provides a single mechanism for all applications, usable by ordinary (non-administrator) users, that can meet the varying data access requirements of applications. DCAC also meets access control requirements that go beyond sandboxing, like user-controlled, fine-grained sharing. A key contribution of DCAC is that it combines the models used by users (file access control and sharing) and administrators (creating sandboxes).

Capsicum has a daemon, Casper, which provides services to sandboxed processes; in DarpaBrowser [25], confined code can access resources in the system via Powerboxes, which is controlled by user interface interactions. These techniques are also applicable to DCAC, providing confined processes an alternative path to reach privileged resources without escalation.

Flexible policy specification. SELinux [16] and AppArmor [7] aim to provide comprehensive policies for the resources that applications can access. This comprehensiveness can lead to usability problems: SELinux is notoriously difficult to use [22]. Both of them are only manageable by administrators and have difficulty accommodating situations where policies for one application vary per user. DCAC is configurable on a per-user basis.

eXtensible Access Control Markup Language (XACML) [5] is an XML-based format for defining access control. While flexible, it relies on XML manipulations (e.g., XPath queries) that are unsuited for use in frequent latency-sensitive operations within an OS.

POSIX ACLs [4] allow for users to define expressive access control lists for permissions. However, these have important limitations, like the inability to support user-defined groups without explicitly putting all group members in the ACL of every file that has group permission (which in turn makes group membership updates difficult). DCAC can accommodate user-defined groups.

The Andrew File System [12] (AFS) supports flexible, per-directory ACLs, and allows users to create groups under their own administration. In comparison, DCAC is not restricted to a specific file system or IPC mechanism, and supports more general usage due to its attribute-based model.

Application- and user-defined access control. Several prior systems allow program-controlled subdivision of users into further users for finer-grained protection [10, 14, 23]. These systems still label processes by one user, and as a result are less flexible than DCAC. Additionally, these systems do not store information about the user hierarchy in a way that easily allows shared use in a network filesystem: UserFS [14] requires synchronization of unrelated applications on a global database of all users. By contrast, DCAC attributes are strings that self-describe where they belong in the attribute hierarchy.

DCAC is inspired by attribute-based access control, proposed as part of InkTag [11]. DCAC generalizes the approach to a trusted OS and makes it coexist with existing access control.

New security models. Decentralized Information Flow Control (DIFC) [9, 15, 17, 21, 29] systems modify access privileges based on the information that applications have accessed. DIFC-enforcing systems may provide stronger security guarantees than DCAC.

Systems that use radically different security models require that developers adapt the logic of their code to work in these models. While DCAC may require code changes to applications, we expect they will be less significant because DCAC's core concept, the attribute, is implemented as a file descriptor, and is easily mapped onto users and groups, concepts that are familiar to developers and likely reflected in their code. Enforcing new security models can require extensive global OS modification, whereas DCAC's changes fit mostly within the existing LSM framework.

9. Conclusion

OS-level support for application-defined principals makes DCAC usable and flexible enough to solve modern access control problems. DCAC decentralizes privilege and policy specification, improves application security, and supports distributed operation.

References

[1] GNU Zlib : List of security vulnerabilities. `http://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-1820/GNU-Zlib.html`.

[2] National Vulnerability Database: CVE-2012-4681. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-4681`.

[3] National Vulnerability Database: CVE-2013-0422. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-0422`.

[4] POSIX 1003.1e Draft (Security APIs). `http://users.suse.com/~agruen/acl/posix/Posix_1003.1e-990310.pdf`.

[5] eXtensible Access Control Markup Language (XACML) Version 3.0. `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cs02-en.html`, August 2012. OASIS Committee Specification 02.

[6] Barth, A., Jackson, C., Reis, C., and Google Chrome Team. The security architecture of the chromium browser. Tech. rep., Google Inc., 2008.

[7] Bauer, M. Paranoid penguin: an introduction to novell AppArmor. *Linux Journal* (2006).

[8] The Chromium Project: Design Documents: OS X Sandboxing Design. `http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design`.

[9] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., and Morris, R. Labels and event processes in the Asbestos operating system. In *SOSP* (2005).

[10] Friberg, C., and Held, A. Support for discretionary role based access control in ACL-oriented operating systems. In *Proceedings of the second ACM workshop on Role-based access control* (1997).

[11] Hofmann, O. S., Dunn, A. M., Kim, S., Lee, M. Z., and Witchel, E. InkTag: Secure applications on an untrusted operating system. In *ASPLOS* (2013).

[12] Howard, J. H., et al. *An overview of the Andrew File System*. Carnegie Mellon University, Information Technology Center, 1988.

[13] Kerrisk, M. Namespaces in operation, part 1: namespaces overview. `https://lwn.net/Articles/531114/`, Jan. 2013.

[14] Kim, T., and Zeldovich, N. Making Linux protection mechanisms egalitarian with UserFS. In *USENIX Security* (2010).

[15] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information flow control for standard OS abstractions. In *SOSP* (2007).

[16] Loscocco, P., and Smalley, S. D. Meeting critical security objectives with security-enhanced linux. In *Proceedings of the Ottawa Linux Symposium* (2001), pp. 115–134.

[17] Myers, A. C., and Liskov, B. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM) 9*, 4 (2000), 410–442.

[18] XATTR protocol patch for NFSv3. `http://namei.org/nfsv3xattr`.

[19] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating system transactions. In *SOSP* (2009).

[20] Provos, N., Friedl, M., and Honeyman, P. Preventing privilege escalation. In *USENIX Security* (2003).

[21] Roy, I., Porter, D. E., Bond, M. D., McKinley, K. S., and Witchel, E. Laminar: Practical fine-grained decentralized information flow control. In *PLDI* (2009).

[22] Schreuders, Z. C., McGill, T., and Payne, C. Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM. *TISSEC* (September 2011).

[23] Snowberger, P., and Thain, D. Sub-identities: Towards operating system support for distributed system security. Tech. rep., University of Notre Dame, 2005.

[24] Sun Microsystems, Inc. RFC 1813 - NFS: Network File System Version 3 Protocol Specification. IETF Network Working Group, 1995.

[25] Wagner, D., and Tribble, D. A Security Analysis of the Combex DarpaBrowser Architecture. *Online at:* `http://www. combex. com/papers/darpa-review` (2002).

[26] Watson, R. N. A decade of os access-control extensibility. *Communications of the ACM* (2013).

[27] Watson, R. N. M., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical Capabilities for UNIX. In *USENIX Security* (2010).

[28] Wright, C., Cowan, C., Morris, J., Smalley, S., and Kroah-Hartman, G. Linux security modules: General security support for the Linux kernel. In *USENIX Security* (2002).

[29] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. Making information flow explicit in HiStar. In *OSDI* (2006).

MiniBox: A Two-Way Sandbox for x86 Native Code

Yanlin Li *CyLab/CMU*

Jonathan McCune *CyLab/CMU, Google Inc.*

James Newsome *CyLab/CMU, Google Inc.*

Adrian Perrig *CyLab/CMU*

Brandon Baker *Google Inc.*

Will Drewry *Google Inc.*

Abstract

This paper presents MiniBox, the first two-way sandbox for x86 native code, that not only protects a benign OS from a misbehaving application, but also protects an application from a malicious OS. MiniBox can be applied in Platform-as-a-Service cloud computing to provide two-way protection between a customer's application and the cloud platform OS. We implement a MiniBox prototype running on recent x86 multi-core systems from Intel or AMD, and we port several applications to MiniBox. Evaluation results show that MiniBox is efficient and practical.

1 Introduction

Platform-as-a-Service (PaaS) is one of the most widely commercialized forms of cloud computing. In 2012, 1 million active applications were running on Google App Engine [14]. On PaaS cloud computing, it is critical to protect the cloud platform from the large number of *untrusted* applications sent by customers. Thus, a virtualized infrastructure (e.g., Xen [7]) and sandbox (e.g., Java sandbox [19]) are deployed to isolate customers' applications and protect the guest OS. However, security on PaaS is not only a concern for cloud providers but also a concern for cloud customers. As shown in Figure 1-A, current sandbox technology provides only one-way protection, which protects the OS from an *untrusted* application. The security-sensitive Piece of Application Logic (PAL) is completely exposed to malicious code on the OS. Also, current sandboxes expose a large interface to *untrusted* applications, and may have vulnerabilities that malicious applications can exploit.

In this paper, we rethink the security model of PaaS cloud computing and argue that a two-way sandbox is desired. The two-way sandbox not only protects a benign OS from a misbehaving application (*OS protection*) but also protects an application from a malicious OS (*application protection*). Researchers have explored several approaches for either protecting the OS from an *untrusted* application [16, 23, 25, 48] or protecting security-sensitive applications (or security-sensitive PALs) from a malicious OS [6, 10, 11, 12, 13, 15, 18, 21, 24, 31, 32, 38, 39, 40, 47]. Unfortunately, none of these schemes provides two-way protection, and many challenges remain to design a two-way sandbox.

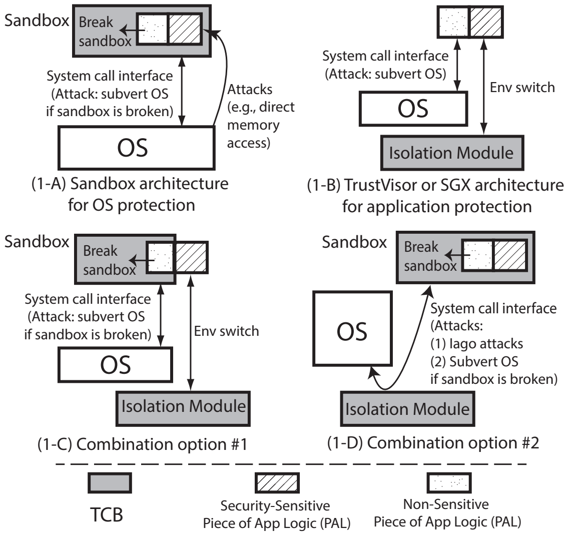

Sandbox
Break sandbox
System call interface (Attack: subvert OS if sandbox is broken)
Attacks (e.g., direct memory access)
OS
(1-A) Sandbox architecture for OS protection
System call interface (Attack: subvert OS)
Env switch
OS
Isolation Module
(1-B) TrustVisor or SGX architecture for application protection
Sandbox
Break sandbox
System call interface (Attack: subvert OS if sandbox is broken)
Env switch
OS
Isolation Module
(1-C) Combination option #1
Sandbox
Break sandbox
OS
System call interface (Attacks: (1) Iago attacks (2) Subvert OS if sandbox is broken)
Isolation Module
(1-D) Combination option #2
TCB
Security-Sensitive Piece of App Logic (PAL)
Non-Sensitive Piece of App Logic (PAL)

Figure 1: Sandbox architecture, TrustVisor or Intel SGX architecture, and combination options.

TrustVisor [31] and Intel Software Guard Extensions (Intel SGX) [4, 17, 20] are examples of systems that provide efficient memory space isolation mechanisms to protect a security-sensitive PAL from a malicious OS (Figure 1-B). On TrustVisor or Intel SGX, memory access from the OS to the security-sensitive PAL or from the security-sensitive PAL to the OS is disabled by an isolation module, which is a hypervisor (on TrustVisor) or CPU hardware extensions (on Intel SGX). However, the non-sensitive PAL is not isolated from the OS, and the non-sensitive PAL may contain malware that can compromise the OS.

Google Native Client (NaCl) [48] and Microsoft Drawbridge [16, 36] are examples of application-layer one-way sandboxes for native code. We found that combining an application-layer sandbox and an efficient memory space isolation mechanism is promising for the two-way sandbox design. However, it it not straightforward. Figure 1-C and 1-D show two combination options. In option #1, the security-sensitive PAL runs in an isolated memory space while a sandbox confines the non-sensitive PAL. However, in this design application developers need to split the application into security-sensitive and non-sensitive PALs, requiring substantial porting effort. In option #2, the sandbox is included inside the isolated memory space to avoid porting. The isolation module forwards system calls (from the sandbox) to the OS. However, there are several issues with this option. First, because the sandbox is complex and exposes a large interface to the application, a malicious application may exploit vulnerabilities in the sandbox and in turn subvert the OS. Second, a malicious OS may be able to compromise the application through Iago attacks [9]. In Iago attacks, a malicious OS can subvert a protected process by returning a carefully chosen sequence of return values to system calls. For instance, if a malicious OS returns a memory address that is in the application's stack memory for an *mmap* system call, sensitive data (e.g., a return address) in the stack may subsequently be overwritten by the mapped data. Finally, because the OS is isolated from the sandbox and the application, it is challenging to support the application execution in an isolated memory space. Thus, both options have obvious shortcomings and we shall not choose them for the two-way sandbox design.

In this paper, we present MiniBox, the first two-way sandbox for x86 native applications. Leveraging a hypervisor-based memory isolation mechanism (proposed by TrustVisor) and a mature one-way sandbox (NaCl), MiniBox offers efficient two-way protection. MiniBox splits the NaCl sandbox into OS protection modules (software modules performing OS protection) and service runtime (software modules supporting application execution), runs the service runtime and the application in an isolated memory space (Section 4.1), and exposes a minimized and secure communication interface between the OS protection modules and the application (Section 4.2). MiniBox also splits the system call interface available to the isolated application as sensitive calls (the calls that may cause Iago attacks) and non-sensitive calls (the calls that cannot cause Iago attacks), and protects the application against Iago attacks by handling sensitive calls inside the service runtime in the isolated memory space (Section 4.3). MiniBox also provides secure file I/O for the application (Section 4.3.4). Using a special toolchain, application developers can concentrate on application development with small porting effort (Section 6). We implement a MiniBox prototype based on the Google Native Client (NaCl) [48] open source project and the TrustVisor hypervisor [31, 41] (Section 5), and port several applications to MiniBox. Evaluation results show that MiniBox is practical and provides an efficient execution environment for isolated applications (Section 6).

Contributions.

1. We design, implement, and evaluate MiniBox, the first attempt toward a practical two-way sandbox for x86 native applications.
2. MiniBox demonstrates it is possible to provide a minimized and secure communication interface between OS protection modules and the application to protect against each other.
3. MiniBox demonstrates it is possible to protect against Iago attacks, and provide an efficient execution environment with secure file I/O for the application.

2 Background

2.1 TrustVisor

TrustVisor [31] is a minimized hypervisor that isolates a PAL from the rest of the system and offers efficient trustworthy computing abstractions (via a μTPM API) to the isolated PAL with a small TCB. TrustVisor isolates the memory pages containing itself and any registered PALs from the guest OS and DMA-capable devices by configuring nested page tables and the IO Memory Management Unit (IOMMU). TrustVisor exposes hypercall interfaces for applications in the guest OS to register and unregister a PAL. When a PAL is *registered*, information including the memory pages of the PAL is passed to TrustVisor. TrustVisor configures nested page tables to isolate the memory pages of the PAL from the guest OS. TrustVisor is booted using the Dynamic Root of Trust for Measurement mechanism [5] available on commodity x86 processors. The chipset computes an integrity measurement (cryptographic hash) of the hypervisor and extends the resulting hash into a Platform Configuration Register (PCR) in the Trusted Platform Module (TPM). TrustVisor computes an integrity measurement for each registered PAL, and extends that measurement result into the PAL's μTPM instance. The TPM Quote from the hardware TPM and the μTPM Quote from the PAL's μTPM instance comprise the complete chain of trust for remote attestation.

2.2 Google Native Client

Google Native Client (NaCl) [48] is a sandbox for x86 native code (called Native Module) using Software Fault Isolation (SFI) [30, 42]. To guarantee the absence of privileged x86 instructions that can break out of the SFI sandbox in a Native Module, a validator in NaCl reliably disassembles the Native Module and validates the disassembled instructions as being safe to execute. NaCl provides a simple service runtime including a context switch function and a system call dispatcher to support the execution of a Native Module. On 32-bit x86, the service runtime and the Native Module are isolating using the CPU's segmentation mechanism [22]. NaCl simulates system calls for a Native Module using a *Trampoline Table and Springboard*. There is a Trampoline Table in each Native Module, and a 32-byte entry in the Trampoline Table for each supported system call. For each system call, the Google NaCl toolchain ensures that control transits to one of the entries in the Trampoline Table, instead of to a traditional system call. The Trampoline Table entries switch the active data and code segments, and jump to the context switch function in NaCl. The context switch function transfers control to the system call dispatcher in NaCl. The system call dispatcher exposes only a subset of the OS system call interface to the Native Module, sanitizes the system call parameters, conducts access control to constrain the file access of the Native Module, and finally calls the corresponding handler in the OS. The *Springboard* performs the inverse of the control transitions in the Trampoline Table entries.

3 Assumptions and Attacker Model

Assumptions. We assume that the attacker cannot conduct physical attacks against the hardware units (e.g., CPU and TPM). We assume that the attacker cannot break standard cryptographic primitives and that the TCB of MiniBox is free of vulnerabilities. For application protection, we also assume that the application does not have any memory safety bugs (e.g., buffer overflows) or insecure designs. One example of the insecure designs is that an application seeds a pseudo-random number generator by the return value of a system call handled by the untrusted OS. It is the developer's responsibility to take measures to eliminate memory safety bugs or insecure designs. For OS protection, we assume that the system call interface that the OS protection modules expose to the application (a subset of the OS system call call interface) is free of vulnerabilities, and that the OS does not have concurrency vulnerabilities [43] in system call wrappers.

Attacker Model For Application Protection. We assume that the attacker can execute arbitrary code on the OS. For example, the attacker may compromise and control the OS, and then attempt to tamper with the protected application by accessing the application memory contents or handling the system calls of the application in malicious ways (Iago attacks). The attacker may attempt to inject malicious code into the application binary or into the service runtime binary before the application runs in an isolated memory space *without being detected*. The attacker may subvert DMA-capable devices on the platform in an attempt to modify memory contents through DMA. The attacker may also attempt to access security-sensitive files of the application. However, we do not prevent denial of service attacks. Finally we do not prevent side-channel attacks [51].

Attacker Model For OS Protection. The untrusted application may attempt to subvert the hypervisor or break out of the hypervisor-based memory isolation. The application may also attempt to read or modify sensitive files that do not belong to the application on the system. The application may attempt to subvert the OS by making arbitrary system calls with carefully-chosen parameters.

4 System Design

4.1 MiniBox Architecture

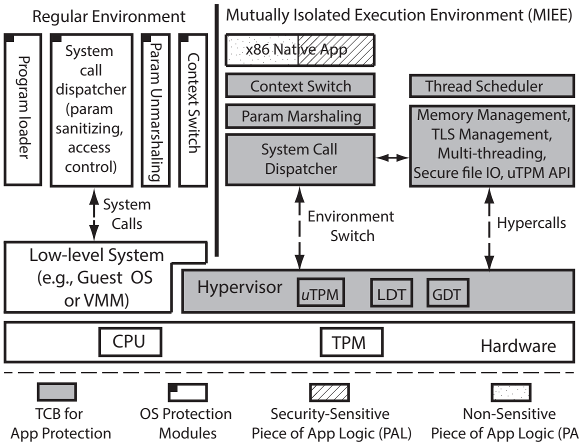

Regular Environment
Mutually Isolated Execution Environment (MIEE)
Program loader
System call dispatcher (param sanitizing, access control)
Param Unmarshaling
Context Switch
x86 Native App
Context Switch
Thread Scheduler
Param Marshaling
Memory Management, TLS Management, Multi-threading, Secure file IO, uTPM API
System Call Dispatcher
System Calls
Environment Switch
Hypercalls
Low-level System (e.g., Guest OS or VMM)
Hypervisor
μTPM
LDT
GDT
CPU
TPM
Hardware
TCB for App Protection
OS Protection Modules
Security-Sensitive Piece of App Logic (PAL)
Non-Sensitive Piece of App Logic (PAL)

Figure 2: MiniBox System Architecture.

Figure 2 shows the MiniBox architecture. As shown in this figure, a hypervisor underpins the system. The hypervisor sets up the two-way memory space isolation between the Mutually Isolated Execution Environment (MIEE) and the regular environment, and creates a μTPM instance for the MIEE.

On MiniBox, the hypervisor and a service runtime in the MIEE comprise the runtime TCB for application protection. In the MIEE, beyond the x86 native application, a service runtime is included, containing: a context switch module that stores and switches thread contexts between the application and the service runtime; a system call dispatcher that distinguishes between non-sensitive and sensitive calls, calls handlers in the MIEE for sensitive calls, or invokes the parameter marshaling module for non-sensitive calls; a parameter marshaling module that prepares parameter information for non-sensitive calls (for the hypervisor); system call handlers for handling sensitive calls; and a thread scheduler that schedules the execution of multiple threads comprising an application. In sensitive call handlers, the service runtime supports dynamic memory management, thread local storage management, multi-threading management, secure file I/O, and μTPM API.

On MiniBox, the OS protection modules include a user-level program loader, a context switch module, a parameter unmarshaling module, and a system calls dispatcher in the regular environment. In the regular environment, the user-level program loader sets up the MIEE and loads the application into the MIEE; the context switch module stores and restores the thread context of the regular environment during environment switches between the regular environment and MIEE; the parameter unmarshaling module unmarshals system call parameters; and the system call dispatcher confines the system call interface exposed to the application (allowing only a subset of the OS system calls), sanitizes the system call parameters, conducts access control to constrain the file access of the application, and forwards the non-sensitive system calls to corresponding handlers in the regular environment.

Finally, MiniBox adopts TrustVisor's integrity measurement (recall Section 2.1) to enable a remote verifier to verify the integrity of the hypervisor, the service runtime, and the isolated application. In this way, MiniBox prevents adversaries from injecting malicious code into the hypervisor, the service runtime or the application before the memory isolation is established without being detected. *This is also the reason that the program loader is not in the TCB for application protection.*

4.2 Communication Interfaces

The MiniBox hypervisor exposes a small interface to the rest of the system. MiniBox minimizes and secures the communication interface between OS protection modules and the application to protect against each other.

Hypervisor Interface. Other than passing system call information between the MIEE and the regular environment, the hypervisor exposes a small interface (i.e., only several hypercalls) to the rest of the system. Thus, assuming the small hypercall interface is free of vulnerabilities, malicious code in the rest of the system cannot compromise the hypervisor or break out of the hypervisor-based memory isolation.

Minimizing Communication Interface. On MiniBox, the communication interface between OS protection modules and the application consists of only the program loader and the system call interface. Because privileged instructions cannot break out of the hypervisor-based memory isolation, the NaCl validator (that validates that the application binary does not contain privileged instructions) is not included in MiniBox, *which significantly reduces the interface exposed to the application*. Without the validator, privileged instructions in the application can break out of the segmentation-based isolation and compromise the service runtime in the MIEE. However, a malicious service runtime in the MIEE cannot break out of the hypervisor-based memory isolation.

Secure Communication. On MiniBox, the hypervisor is the only communication channel between the regular environment and the MIEE. Each non-sensitive system call causes environment switches between the MIEE and the regular environment. For each environment switch from the MIEE out to the regular environment, the parameter marshaling module in the MIEE updates the parameter information of the system call that will be used by the hypervisor for copying parameters between the two environments. However, the parameter marshaling module in the MIEE cannot specify where the parameters will be stored in the regular environment. The hypervisor copies the system call parameters to a parameter buffer in the regular environment, and constrains the total data size of system call parameters (to prevent buffer overflow attacks). In this way, malicious code in the MIEE cannot overwrite critical data (e.g., stack contents) in the regular environment. To prevent a misbehaving application from sending arbitrary system call parameters to the regular environment, the system call dispatcher in the regular environment checks the system call parameters before sending them to the OS. For example, the system call dispatcher checks the value of every pointer parameter and guarantees that it is safe to access the memory space the parameter points to. If a check fails, the system call dispatcher returns an error code without calling the corresponding system call handler.

After the system call is handled, the system call dispatcher copies return values to the parameter buffer in the regular environment and triggers the environment switch back to the MIEE. When MiniBox switches from the regular environment *back to* the MIEE, the hypervisor uses the same parameter information specified by the MIEE to copy parameters from the parameter buffer in regular environment to the MIEE. This prevents malware in the regular environment from attempting to compromise MIEEs by manipulating parameter information.

4.3 Service Runtime

4.3.1 Dynamic Memory Management

MiniBox supports three system calls (`sysbrk`, `mmap`, and `munmap`) to provide dynamic memory management for the application running inside the MIEE. To prevent the OS from returning arbitrary memory addresses for the `sysbrk` or `mmap` system calls (Iago attacks) or removing arbitrary data memory pages from the MIEE, memory management system calls are handled inside the MIEE.

Design. One naive design is pre-allocating and registering a large amount of data memory in the MIEE as data memory for the application. This design has low execution time overhead, but it wastes memory and is inflexible. Another design is allowing the hypervisor to allocate memory pages as the application's data memory. However, the MiniBox hypervisor does not support swapping of memory pages to disk, and cannot be sure that pages marked as unused by the guest OS are actually present in memory. To resolve this issue, we design the system call handlers that request more data memory (i.e., `sysbrk` and `mmap`) in two modules: one in each of the isolated and regular environments. When the application requests more data memory but the requested data memory is not in the MIEE, the system call handler in the MIEE calls the module in the regular environment that allocates the memory page(s) and writes zero to them to ensure that the new memory page(s) are loaded into physical memory, and then returns to the handler inside the MIEE. The system call handler inside the MIEE then makes a hypercall to the hypervisor to add the new memory page(s) to the MIEE. The `munmap` handler inside the MIEE makes a hypercall to unregister memory from the MIEE.

Security Protection. To prevent Iago attacks caused by `mmap` or `sysbrk`, the hypervisor checks that the newly registered pages are not already registered to the MIEE (so that the malicious OS cannot overwrite stack contents of the application in the MIEE). To prevent leakage of sensitive data in either direction, the MiniBox hypervisor *zeroes* memory pages during registration and unregistration. To prevent a misbehaving or malicious application from adding privileged data pages (e.g., kernel pages) into MIEE, the hypervisor checks that the newly registered pages are user-level memory pages that are in ring 3, and correspond to the same OS process that originally registered the MIEE. Presently MiniBox does not allow additional memory to be mapped as executable, as this represents a significant increase in attack surface. Thus, the hypervisor checks that the requested memory pages are data pages that are not executable. In *data* memory page unregistration, the hypervisor checks that the unregistered memory pages are data pages that are already registered to the MIEE.

4.3.2 Thread Local Storage Management

Background. On 32-bit Linux, the native code on vanilla NaCl stores the memory address of its Thread Local Storage (TLS) as the base address of a segment descriptor in the Local Descriptor Table (LDT) [22]. During program initialization or when a new thread is created, `tls_init` system call initializes the TLS base address and updates the appropriate LDT entry. During execution, the `tls_get` system call is frequently called to get the TLS base address.

Design. Because the TLS and LDT represent critical configuration data, MiniBox handles the `tls_init` and `tls_get` entirely within the MIEE. The MiniBox hypervisor creates an LDT instance for each MIEE and supports a hypercall interface to the MIEE to handle `tls_init` system call. MiniBox supports caching the latest TLS address inside the MIEE, so that the `tls_get` handler can quickly return the latest TLS base address to the application without calling the hypervisor.

4.3.3 Multi-threading

Background. NaCl applies a 1:1 thread model (i.e., creating a kernel thread for each Native Module user-level thread) and uses the OS to handle thread-related system calls (e.g., thread synchronization system calls) and schedule the execution of Native Module threads.

Design. If MiniBox applies the same multi-threading mechanism, the OS controls the thread context of the application threads. A malicious OS could break the Control Flow Integrity (CFI) [1, 2, 3] of the isolated application by changing the thread context. Also, when the OS handles all thread synchronization system calls, a malicious OS could break the application CFI by arbitrarily changing application thread states. To protect the application thread context from being accessed by the OS, MiniBox can store the thread context in the MIEE and never leak it out of the MIEE. Also, the service runtime in the MIEE can verify the thread synchronization results by duplicating all supported thread synchronization system call handlers. In this design, all thread context and the application CFI are protected from a malicious OS. However, the complexity of this design is comparable to implementing the multi-threading operations within the MIEE. Also, if thread-related system calls are handled by the OS, the environment switches caused by thread-related system calls will increase the overhead of application execution in the MIEE. Thus, *to reduce execution overhead and avoid duplicated operations, MiniBox supports multi-threaded application execution via a user-level multi-threading mechanism entirely within the MIEE. System calls to create, exit and synchronize threads are handled in the MIEE.*

Thread Scheduler. MiniBox provides a thread scheduler to schedule the thread execution of the application in the MIEE. The thread scheduler is invoked each time there is a call from an entry of the *Trampoline Table* (recall Section 2). After the call is handled, control returns to the *thread scheduler* inside the MIEE before the context switch module is invoked. The scheduler checks the state of each thread, and schedules the execution of runnable threads using a round-robin algorithm. The thread scheduler finally calls the context switch module, which resumes the execution of the scheduled thread.

4.3.4 Secure File I/O

On MiniBox, the application running in the MIEE still needs to access the file system in the regular environment, so the file system calls are forwarded to the OS. However, to protect the file contents and metadata of an isolated application, MiniBox supports secure file I/O for applications running in the MIEE through five system calls: `secure_write`, `secure_read`, `secure_open`, `secure_close`, `create_siokey`. The five system calls are handled in the MIEE.

Confidentiality and Integrity. `secure_write` encrypts the data written by the application (with a symmetric secret key) and sends the encrypted data to the general file I/O, while `secure_read` decrypts the data and returns the decrypted data to the application in the MIEE. In `secure_write` and `secure_read`, the data is written or read by a chain of blocks of a constant size. To protect the integrity of file contents and file metadata, including file name and path, a hash tree is constructed and computed over the blocks of file contents and file metadata in the MIEE (this approach has been demonstrated in the Trusted Database System [28], VPFS [45] and jVPFS [46]). A HMAC of the master hash is computed in the MIEE and stored at the end of the file (as file contents). When a file created by secure file I/O is opened, `secure_open` reads the HMAC and verifies the integrity of the file contents and metadata by reconstructing the hash tree. `secure_open` stores the hash tree in the MIEE. When a data block is read, `secure_read` verifies the integrity of the data block based on the stored hash tree. When file contents are modified, `secure_write` updates the hash tree stored in the MIEE. When a file is closed, `secure_close` recomputes the master hash and the HMAC, and stores the updated HMAC at end of the file. This allows the integrity of file contents and file metadata to be verified. The attacker cannot remove, add, or replace data blocks in the file because any changes will invalidate the HMAC. The attacker cannot replace the file with other files that are created by the same application running in the MIEE either because file metadata is also verified.

Rollback Prevention (Freshness). MiniBox adds a counter in each HMAC computation to guarantee freshness of files stored through the secure file I/O. The counter is sealed by the μTPM. Because the μTPM cannot provide freshness for sealed contents, the integrity of the counter is measured every time the same application runs in the MIEE (the measurement result is extended into μPCR for remote attestation). This allows a verifier to verify the freshness during remote attestation.

Key Management. Before using secure file I/O, the application running in the MIEE must call `create_siokey` to create the secret keys used in secure file I/O (i.e., a symmetric encryption key and a HMAC key). The application specifies the file name and file path for storing the keys when calling `create_siokey`. `Create_siokey` first checks if the file already exists. If not, `create_siokey` creates new secret keys, seals the secret keys with the current μPCR values. Then it stores the sealed secret keys in the file, and returns the key ID to application. If the file already exists (i.e., keys are already created), `create_siokey` reads the sealed keys from the untrusted file system, unseals the keys and returns the key ID to the application.

Access Control and Migration. Because the secret keys are sealed with the current μPCR (i.e., the integrity measurement of the application), the sealed keys can only be unsealed by the μTPM when the same application runs in the MIEE. Thus, any data encrypted through secure File I/O can only be decrypted and verified when the same application runs in the MIEE. To share the sensitive files with other applications running in the MIEE (e.g., an updated version of the application), the application can seal the secret keys with the integrity measurement result of other applications, and share the sealed keys to other applications. Then, other applications running in the MIEE can unseal the secret keys (using `create_siokey`) and access the secret files.

Cache Buffer. On MiniBox, environment switches between the MIEE and the regular environment cause high overhead in file I/O (Section 6). To reduce the number of environment switches, MiniBox creates a cache buffer in the MIEE for each opened file descriptor. Both general file I/O and secure file I/O benefit from the cache buffer because the number of environment switches is reduced.

4.4 MIEE Preemption and Scheduling

As described in Section 4.3.3, MiniBox does not preempt an application thread running in the MIEE. However, if an application thread is in an endless loop, the thread will not freeze the entire system because the MIEE is preemptive on MiniBox. When the system switches into a MIEE, the hypervisor starts a timer for the MIEE and preempts the code execution in the MIEE when the timer expires. After preempting the MIEE, the hypervisor stores the MIEE context and transfers control to the regular environment by simulating a special system call (i.e., *MIEE_sleep*). The *MIEE_sleep* handler sleeps for a while and then calls the hypervisor to resume the code execution in the MIEE. In this way, the hypervisor transfers the control to the OS, which can schedule the execution of other processes. When multiple MIEEs are registered (one MIEE in each process), the OS can implicitly schedule the execution of multiple MIEEs by scheduling process execution. However, the question is how much CPU time should be assigned to each MIEE by the hypervisor. One design is that the hypervisor exposes a hypercall interface to the regular environment and the MIEE to enable the OS and the isolated application in the MIEE to configure the MIEE process priority. The hypervisor assigns CPU time to each MIEE based on the MIEE process priority.

4.5 Exceptions, Interrupts, and Debugging

Exceptions and Interrupts. While the code in a MIEE is running, the processor cannot access exception and interrupt handlers in the OS. Thus, the hypervisor is configured to intercept exceptions (e.g., *segmentation fault*, *invalid opcode*) and Non-Maskable Interrupts (NMIs) when system runs in a MIEE. Maskable interrupts are disabled when system runs in a MIEE. When NMIs happen, the hypervisor handles NMIs and resumes the code execution in the MIEE. When an exception happens, the hypervisor first checks whether the exception is because the application in the MIEE needs more stack pages. If so, the hypervisor calls a module in the regular environment to allocate more data pages as stack pages, adds the stack pages into the MIEE, and resumes the code execution in the MIEE. If not, the hypervisor terminates the code execution in the MIEE by simulating an *Exit* system call. The *Exit* call is forwarded to the program loader, which unregisters the MIEE from the hypervisor via hypercall.

Debugging. Though the MiniBox execution environment is compatible with NaCl's, the NaCl debugging tool for application development cannot be directly used on MiniBox because on MiniBox the OS cannot access the memory contents in the MIEE. However, MiniBox can be configured in a debugging mode, in which the hypervisor functionalities are disabled, and an application layer module passes parameters between the two environments. In debugging model, memory management and TLS management calls are handled by the OS. In this way, the memory isolation is disabled and application developers can use the NaCl debugging tool for MiniBox application development. An alternative way is including the NaCl debugging tool in the MIEE and supporting an interface to access the debugging tool from the regular environment. In this way, the developers can debug the application when the memory isolation is enabled.

5 Implementation

We implement a MiniBox prototype running on recent x86 multi-core systems from Intel or AMD, with 32-bit Ubuntu 10.04 LTS as the guest OS. This section describes the MiniBox implementation in details.

5.1 Hypervisor

The implementation of the MiniBox hypervisor is based on the public implementation of TrustVisor hypervisor (version 0.1.2) [31, 41] with support for multi-core and both AMD and Intel processors. We changed the parameter marshaling implementation [26] and added a hypercall interface for handling sensitive system calls. We added code to create new Global Descriptor Table (GDT) [22] entries and instantiate an LDT for every MIEE, and added code to handle GDT- and LDT-related operations. The original implementation of TrustVisor hypervisor has 14414 source lines of code (SLoC), computed using the sloccount tool[1]. Our implementation adds an additional 691 SLoC.

5.2 Program Loader and Service Runtime

We implement the user-level program loader, the service runtime in the MIEE, the context module and the system call dispatcher in the regular environment based on the Google Native Client (NaCl) open source project (SVN revision 7110). We have focused our work on the 32-bit x86 architecture, though there are no fundamental barriers to expanding to 64-bit. In the NaCl source code, we implement code to conduct MIEE registration and unregistration in 299 SLoC. We implement the service runtime in the MIEE within the NaCl source code, adding 3550 SLoC. The secure file I/O module has a large code base (1065 SLoC) because it contains cryptographic primitives for AES and HMAC. The implemented service runtime can be configured in debugging mode for application development (recall Section 4.5).

5.3 System Calls

MiniBox adopts NaCl system call interface to expose a subset of the OS system call interface to the isolated application. MiniBox does not support dynamic code for the application, so NaCl dynamic code system calls are removed on MiniBox. MiniBox extends the NaCl system call interface with μTPM API, network I/O system calls, and secure file I/O calls, supporting a total of 75 system calls for the application (a list of supported system calls is described in [26]). The network I/O system calls are forwarded to the regular environment, because they are treated as part of the untrusted communication channel. Secure communication (e.g., SSL) can be implemented in the application layer to protect the data in network I/O. In the MIEE, the supported thread synchronization system calls include semaphores, mutexes, and condition variables, which have the same functionality as the corresponding POSIX APIs. The secure file I/O calls encrypt/decrypt the data using AES with a 128-bit key in CBC mode and computes HMAC-SHA-1 using a 160-bit key.

6 Evaluation

In this section, we present the evaluations including system call overhead, file I/O overhead, network I/O, and

[1] http://www.dwheeler.com/sloccount/

application performance in the MIEE on MiniBox. Experiments were conducted on a Dell PowerEdge T105 server with a Quad-Core AMD Opteron Processor running at 2.3 GHz with 4 GB memory. The operating system is Ubuntu 10.04 with 32-bit kernel Linux 2.6.32.27. To obtain accurate timing results, the hypervisor does not preempt the MIEE.

Performance Impact. MiniBox hypervisor extends the TrustVisor with hypercall interface and modified parameter marshaling [26], neither of which affects the guest OS performance. Thus, MiniBox hypervisor imposes similar guest overhead to the TrustVisor [41]. Yee et al. [48, 49] presented that the NaCl toolchain can cause significant increase in code size (2% to 57% on SPEC2000 benchmarks), but non-significant impact on performance (on average less than 5% on SPEC2000 benchmarks).

Porting Effort. MiniBox uses the NaCl toolchain with extended API for application development and imposes similar porting efforts to the NaCl. Yee et al. [48, 49] presented that porting an internal implemented H.264 decoders (11K lines of C code) to NaCl requires adding about twenty lines of C code, and porting Bullet[2] to NaCl took only a few hours. Compared to NaCl, MiniBox requires additional porting effort for application protection. For instance, application developers must understand the MiniBox protection mechanisms and avoid insecure application designs (recall Section 3). Application developers must understand the trustworthy computing abstractions exposed to every MIEE, and correctly use them.

6.1 MiniBox Microbenchmarks

System Call Overhead. In the MIEE, non-sensitive system calls are handled in the OS with environment switches while sensitive system calls are handled either in the application layer inside the MIEE or by the hypervisor. The system call overhead in the MIEE was measured, and compared with the corresponding system calls on vanilla NaCl, and MiniBox in debugging model (recall Section 4.5). The evaluation results (Figure 3) show that the non-sensitive system calls (e.g., file operation calls) that involve environment switches on MiniBox are slower than on vanilla NaCl. However, the corresponding system calls on MiniBox in debugging mode have similar performance to those on vanilla NaCl. Thus the overhead of these system calls on MiniBox is mainly caused by environment switches. The sensitive system calls that are handled within the MIEE without any environment switch (e.g., thread synchronization calls) have similar performance to those on vanilla NaCl. The sensitive system calls that involve hypercall and environment switches (e.g., memory management system calls) on MiniBox are slower than on vanilla NaCl.

File I/O. We evaluate the file I/O overhead on MiniBox and compare it to the file I/O on vanilla NaCl and MiniBox in debugging mode. We measure reads & writes of 32B for both general file I/O and secure file I/O. The measurement results (Figure 4) show that when the data is cached in the MIEE (cache-hit), the cache buffer significantly reduces the file I/O overhead for both general file I/O and secure file I/O.

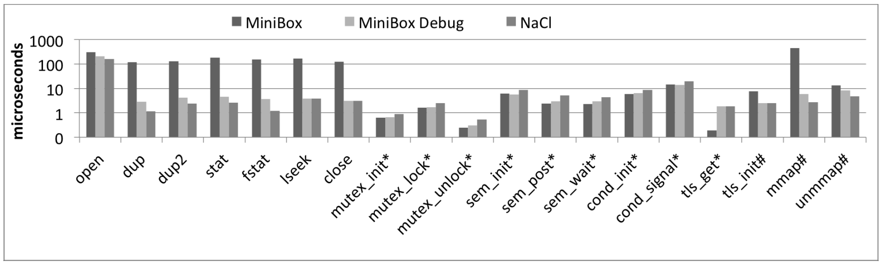
Network I/O. We evaluate the network I/O throughput on MiniBox and compare it to the network I/O throughput on MiniBox in debugging mode and vanilla NaCl. The server runs in the MIEE using MiniBox on the Dell T105 while the client runs on plain Linux on a Dell Optiplex 755 desktop with two Intel Core2 Duo processors running at 2.0 GHz with 2 GB memory. The operating system on the Dell Optiplex machine is Ubuntu 8.04.4 LTS with a 32-bit Linux kernel 2.6.24.30. Both the server and the client connect to a Netgear Gigabit Ethernet Switch using a Gigabit Ethernet Adapter. During each connection, the client sends 16 KB data to the server and we measure the network I/O throughput. The results (Figure 5) show that network I/O on MiniBox is about 10% slower than on vanilla NaCl. *Thus, although the environment switches impose a small overhead on MiniBox, the network throughput remains high.*

6.2 Application Benchmarks

CPU-bound application (AES key search and BitCoin). We measure the performance of CPU-bound applications on MiniBox and compare it to the performance of equivalent applications on vanilla NaCl and MiniBox in debugging mode. We first evaluate *AES key search*, which encrypts a 128-Byte plain-text using a 128-bit key in CBC mode 200,000 times, simulating a AES key search operation. We port CBitCoin [33]), an open source BitCoin implementation to run on MiniBox. We measure the time to construct a BitCoin block, requiring 200,000 SHA-256 computations. The results show that *MiniBox does not add any noticeable overhead (less than 1% [26]) for CPU-bound applications over NaCl.*

I/O-bound application (Zlib). We evaluate the performance of I/O-intensive applications on MiniBox by testing Zlib [27], an open source library used for data compression. Zlib is already ported to run on NaCl as part of the naclports project, and does not require additional porting efforts to run on MiniBox. We measure the time elapsed to read 1 MB of file data from the file system over the general file I/O, and then compress the read data. The file data always misses the cache buffer, so every *read* operation involves an environment switch. The evaluation results (Figure 6) show that because of

[2] http://www.bulletphysics.com



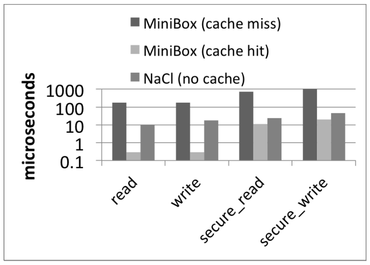
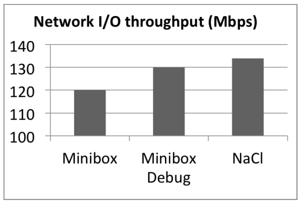
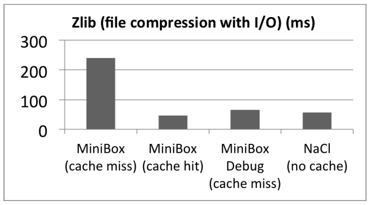
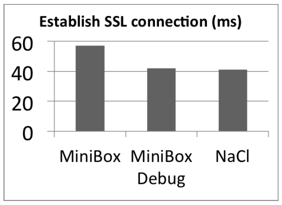


Figure 3: System call benchmarks in *us*. Average of 100 runs and standard deviation is less than 5%. Calls with ∗ are sensitive calls handled inside the MIEE without environment switches. Calls with # are sensitive calls that involve hypercall or environment switches.

Figure 4: File I/O benchmarks in *us*. Average of 100 runs and standard deviation is less than 2%.







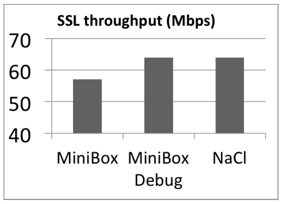


Figure 5: Network I/O benchmarks in *Mbps*. Average of 100 runs and standard deviation is less than 2%.

Figure 6: zlib file compression with file I/O benchmarks in *ms*. Average of 10 runs and standard deviation is less than 2%.

Figure 7: SSL connection benchmarks in *ms*. Average of 10 runs and standard deviation is less than 3%.

Figure 8: SSL throughput benchmarks in *Mbps*. Average of 10 runs and standard deviation is less than 1%.

environment switches, the zlib application on MiniBox is slower than on vanilla NaCl. The slowdown is mainly caused by the environment switches since MiniBox in debugging mode has the same performance as vanilla NaCl. We repeat the measurement on MiniBox while storing the file data in the cache buffer in the MIEE. The zlib application read file data with cache-hit without environment switches. The measurement result shows that the overhead is significantly reduced. *Thus, while file I/O in MiniBox can be expensive in the worst case, we expect that the cache buffer will significantly improve the application performance in practice.*

SSL Server. We port the entirety of OpenSSL [35] (version 1.0.0.e) to run on MiniBox. We also run the SSL server on NaCl by adding socket system call interface on the NaCl. In this experiment, the Dell Optiplex machine serves as the SSL client, and the Dell T105 acts as the SSL server. The SSL client runs on plain Linux while the SSL server runs inside the MIEE on MiniBox. We recorded both the time required to create an SSL connection and the overall SSL throughput. The SSL client sends 16KB of data to the SSL server during each connection. As in previous experiments, both machines connect to a Netgear Gigabit Ethernet Switch via a Gigabit Ethernet Adapter. The results show that MiniBox impose about a 15% overhead to SSL connections (Figure 7) and that SSL throughput on MiniBox has about a 10% slowdown (Figure 8). The overhead is mainly caused by environment switches, since MiniBox in debugging mode has the same performance as NaCl.

7 Related Work

Protecting Applications. Systems aspiring to protect entire applications from a potentially compromised OS have been proposed (e.g., [8, 11, 12, 13, 15, 21, 29, 34, 40, 47]). Most of these schemes mainly focus on protecting application data from malicious code on an operating system and expose sensitive system calls to the untrusted OS, thus making the protected application vulnerable to Iago attacks. InkTag [21] secures applications running on an untrusted OS by verifying that the untrusted OS behaves correctly using a trustworthy hypervisor. It prevents `mmap`-based Iago attacks by verifying memory address invariants. However, in InkTag some other security-sensitive system calls (e.g., thread synchronization and TLS-related calls) are still performed by the untrusted OS without being verified. Proxos [40] splits system calls and forwards sensitive system calls to a trusted private OS to protect applications from an untrusted OS. However, Proxos needs application developers to specify the splitting rule. Baumann et al. [8] proposed to run entire legacy applications in the isolated memory space provided by Intel SGX, and proposed to include a library OS in the isolated memory space to prevent Iago attacks. The proposed protection mechanisms (for application protection) are similar to the mechanisms on MiniBox. Mai et al. [29] proposed mechanisms to prove that the OS implements the application security invariants (e.g., secure storage and memory isolation) correctly. The proposed verification approach is promising for application isolation.

Protecting Security-Sensitive Code. Researchers have explored many systems for isolating sensitive code using virtualization, microkernels, and other low-level mechanisms [6, 18, 31, 32, 38, 40], or by running the code inside trusted hardware [10, 24, 39]. The virtualization-based schemes contain a large TCB. Other schemes either do not enjoy compatibility with a large set of commodity systems or require significant porting effort. TrustVisor [31] and Flicker [32] isolate a PAL from an untrusted OS with a small TCB. However, porting security-sensitive applications on TrustVisor or Flicker requires significant efforts. Nizza [38] also requires developers to perform similar operations to port sensitive applications to Nizza.

Sandbox for x86 Native Code. Google Native Client [48] confines untrusted native code using SFI [30, 42] and enables developers to port native code as web applications. Drawbridge [16, 36] isolates an application in a picoprocess and provides a library OS to the isolated application. However, Native Client and Drawbridge provide only one-way protection. TxBox [23] confines an untrusted application by executing the application in a system transaction and conducting security check. MBox [25] protects the host file system from an untrusted application by exposing a virtual file system on top of the host file system for the application. Capsicum [44] supports capability-sandbox for applications on UNIX-like OS (e.g., FreeBSD). It focuses on application compartmentalization and fine-grained access control. Systrace [37] improves the host OS security by confining the program privilege using a configurable system call policy. The protection mechanisms provided by MBox, Capsicum and Systrace can be applied on MiniBox as part of the OS protection modules.

8 Limitations and Future Work

Application Interface. MiniBox includes the entire application (the security-sensitive and non-sensitive PALs) in the MIEE and does not prevent adversaries from compromising the application through malicious inputs. The application can measure the integrity of critical inputs (known inputs) and extend the results into the μTPM PCR for remote attestation. However, the isolated application may expose a large interface to unknown inputs. Schemes that focus on protecting a security-sensitive PAL [6, 18, 31, 32, 38, 40] can significantly reduce the attack surface by exposing a constrained interface between the security-sensitive PAL and the untrusted OS. On those schemes, the security-sensitive PAL remains secure when the application is compromised by the OS. Thus, for protecting the security-sensitive PAL, MiniBox may expose a larger attack surface to the untrusted OS than schemes that focus on protecting the security-sensitive PAL.

Thread Scheduling. Application developers must consider that MiniBox does not make the scheduler work preemptively (recall Section 4.3.3), and so must always use supported system calls for thread synchronization (e.g., avoid situations where a thread performs busy waiting by watching a global variable in a loop instead of calling a blocking system call). In addition, the application-layer thread scheduler does not support multi-thread parallel computation to improve the performance of threaded applications on multi-core systems. One design is to allow the hypervisor to conduct thread scheduling and to manage the parallel computation on multiple cores, which will significantly increase the hypervisor complexity. As future work, we will investigate how to support parallel computation for a threaded application running inside the MIEE on multi-core systems. However, security-sensitive applications more concerned with a small TCB than performance may prefer not to include code for such complex operations in the hypervisor. To solve this issue, MiniBox can allow the application to configure the hypervisor functionality (e.g., disable the support for multi-thread parallel computation) at registration time, and can boot the hypervisor with the application-preferred configurations.

System Call Interface. Exposing a large system call interface to the application increases the attack surface for OS protection; thus, MiniBox exposes a subset of the OS system call interface to the application to confine the application's operations. However, it will be interesting to investigate how to support the entire OS system call interface on MiniBox. If the entire OS system call interface is supported, statically linked legacy applications may be able to run on MiniBox. As future work, we will examine the OS system call interface, obtain a comprehensive list of sensitive calls, and investigate how to support the entire OS system call interface on MiniBox.

Improving Performance. The hypervisor-based isolation mechanism causes overhead in environment switches. It is expected that the hardware-based isolation mechanism provided by Intel SGX will decrease the environment switch overhead. The *VMFUNC* instruction [22] released on the latest Intel 4th Generation Processor enables software in a guest Virtual Machine to switch nested page tables without a Virtual Machine exit. It is expected that the *VMFUNC* instruction will decrease the environment switch overhead. However, the *VMFUNC* instruction does not switch other critical system configurations (e.g., the GDT or IDT). As future work we will investigate how to perform secure environment switch using the *VMFUNC* instruction.

Supporting Multi-tenant Cloud Platform. The MiniBox hypervisor prototype supports only a single guest OS. There is no fundamental barrier to port MiniBox with a virtual machine monitor like Xen [7] that supports multiple tenants, though doing so increases the TCB size. CloudVisor [50] demonstrates the approach to minimize the TCB on multi-tenant cloud platforms by leveraging nested virtualization technology. Nested virtualization can be added in MiniBox to support multi-tenant cloud platforms. On multi-tenant cloud platforms, the virtual machine (VM) may be constructed, destructed, saved, restored, or migrated. It is critical to protect the MIEE during VM management. The MiniBox hypervisor can encrypt or decrypt the memory contents of MIEEs in VM management, and verify the integrity of the MiniBox hypervisor on other machines to guarantee that MIEEs are only migrated to machines with a verified hypervisor. Also, the MiniBox hypervisor needs to encrypt or decrypt the μTPM instance together with a MIEE in VM management, to make the trustworthy computing abstractions provided to the MIEE transparent to the VM management.

9 Conclusion

MiniBox is a hypervisor-based sandbox that provides two-way protection between x86 native applications and the guest OS. MiniBox protects the guest OS through hypervisor-based memory isolation and OS protection modules. MiniBox significantly reduces the attack surface for both OS protection and application protection by minimizing and securing the interface between OS protection modules and the application, and protects against Iago attacks on the application. The MiniBox design and protection mechanisms are promising for establishing two-way protection on commodity computer systems. In addition, MiniBox significantly decreases the porting effort compared to previous systems for isolating security-sensitive PALs, making MiniBox practical for wide adoption. Thus, we anticipate that MiniBox will be widely adopted on systems where two-way protection is desired (e.g., the PaaS cloud computing platforms).

Acknowledgements

Many thanks to J. Bradley Chen and Haibo Chen for helpful discussion about this work. We thank the anonymous reviewers for their time, attention, and valuable suggestions, and thank Bennet Yee, Kyle Orland, and Steve Matsumoto for improving the writing in the paper.

References

[1] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. CFI: Principles, Implementations, and Applications. In *Proceedings of ACM Conference and Computer and Communications Security* (2005).

[2] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-Flow Integrity Principles, Implementation, and Applications. *ACM Transaction on Information and System Security* (2009), 1 – 40.

[3] ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. A theory of secure control flow. In *Proceedings of Conference on Formal Engineering Methods* (2005).

[4] ABATU, U., GUERON, S., JOHNSON, S. P., AND SCARLATA, V. R. Innovative technology for CPU based attestation and sealing. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[5] ADVANCED MICRO DEVICES. AMD64 architecture programmer's manual: Volume 2: System Programming. AMD Publication no. 24593 rev. 3.14, Sept. 2007.

[6] AZAB, A. M., NING, P., AND ZHANG, X. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of ACM Conference on Computer and Communications Security* (2011).

[7] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of Symposium on Operating Systems Principles* (2003).

[8] BAUMANN, A., PEINADO, M., HUNT, G., ZMUDZINSKI, K., ROZAS, C. V., AND HOEKSTRA, M. Secure execution of unmodified applications on an untrusted host. `http://research.microsoft.com/apps/pubs/default.aspx?id=204758`, 2013.

[9] CHECKOWAY, S., AND SHACHAM, H. Iago attacks: Why the system call API is a bad untrusted rpc interface. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (Mar. 2013).

[10] CHEN, B., AND MORRIS, R. Certifying program execution with secure processors. In *Proceedings of HotOS* (2003).

[11] CHEN, H., ZHANG, F., CHEN, C., YANG, Z., CHEN, R., ZANG, B., YEW, P., AND MAO, W. Tamper-resistant execution in an untrusted operating system using a VMM. Tech. Rep. FDUPPITR-2007-0801, Fudan University, 2007.

[12] 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 International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[13] CHENG, Y., DING, X., AND DENG, R. AppShield: Protecting applications against untrusted operating system. In *Singaport Management University Technical Report, SMU-SIS-13-101* (2013).

[14] DARROW, B. Google App Engine by the numbers. `http://gigaom.com/2012/06/28/google-app-engine-by-the-numbers/`.

[15] DEWAN, P., DURHAM, D., KHOSRAVI, H., LONG, M., AND NAGABHUSHAN, G. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of Spring Simulation Multiconference* (2008).

[16] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation* (2008).

[17] FRANK, M., ILYA, A., ALEX, B., V, R. C., HISHAM, S., VEDVYAS, S., AND R, S. U. Innovative instructions and software model for isolated execution. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[18] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of ACM Symposium on Operating System Principles* (2003).

[19] GONG, L. Java 2 Platform Security Architecture. http://docs.oracle.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html.

[20] HOEKSTRA, M., LAL, R., PAPPACHAN, P., PHEGADE, V., AND DEL CUVILLO, J. Using innovative instructions to create trustworthy software solutions. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy* (2013).

[21] HOFMANN, O., DUNN, A., KIM, S., LEE, M., AND WITCHEL, E. InkTag: Secure applications on an untrusted operating system. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[22] INTEL CORPORATION. Intel 64 and IA-32 architectures software developer's manual volume 3b: system programming guide, part 2. Order Number: 325384-048US, Sept. 2013.

[23] JANA, S., PORTER, D. E., AND SHMATIKOV, V. TxBox: Building secure, efficient sandboxes with system transactions. In *Proceedings of IEEE Symposium on Security and Privacy* (2011),

[24] JIANG, S., SMITH, S., AND MINAMI, K. Securing web servers against insider attack. In *Proceedings of Computer Security Applications Conference* (2001).

[25] KIM, T., AND ZELDOVICH, N. Practical and effective sandboxing for non-root users. In *Proceedings of USENIX Annual Technical Conference* (2013).

[26] LI, Y., PERRIG, A., MCCUNE, J. M., NEWSOME, J., BAKER, B., AND DREWRY, W. MiniBox: A Two-Way Sandbox for x86 Native Code. Tech. Rep. CMU-CyLab-14-001, Carnegie Mellon University, 2014.

[27] LOUP GAILLY, J., AND ADLER, M. zlib open source library. http://www.zlib.net.

[28] MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, W. How to build a trusted database system on untrusted storage. In *Proceedings of USENIX Symposium on Operating System Design & Implementation* (2000).

[29] MAI, H., PEK, E., XUE, H., KING, S. T., AND MADHUSUDAN, P. Verifying security invariants in expressOS. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[30] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Proceedings of USENIX Security Symposium* (2006).

[31] 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 IEEE Symposium on Security and Privacy* (2010).

[32] MCCUNE, J. M., PARNO, B., PERRIG, A., REITER, M. K., AND ISOZAKI, H. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of European Conference on Computer Systems* (2008).

[33] MITCHELL, M., STERLING, A., AND MILLER, A. Cbitcoin open source project. http://code.google.com/p/naclports/.

[34] ONARLIOGLU, K., MULLINER, C., ROBERTSON, W., AND KIRDA, E. PrivExec: Private execution as an operating system service. In *Proceedings of IEEE Symposium on Security and Privacy* (2013).

[35] OPENSSL PROJECT TEAM. OpenSSL. http://www.openssl.org/, May 2005.

[36] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library OS from the top down. *SIGPLAN Not. 46*, 3 (Mar. 2011), 291–304.

[37] PROVOS, N. Improving host security with system call policies. In *Proceedings of USENIX Security Symposium* (2003).

[38] SINGARAVELU, L., PU, C., HÄRTIG, H., AND HELMUTH, C. Reducing TCB complexity for security-sensitive applications. In *Proceedings of European Conference on Computer Systems* (2006).

[39] SMITH, S. W., AND WEINGART, S. Building a high-performance, programmable secure coprocessor. *Computer Networks 31*, 8 (Apr. 1999).

[40] TA-MIN, R., LITTY, L., AND LIE, D. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of ACM Symposium on Operating Systems Principles* (2006).

[41] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., AND DATTA, A. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of IEEE Symposium on Security and Privacy* (2013).

[42] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of ACM Symposium on Operating Systems Principles* (1993).

[43] WATSON, R. N. M. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of USENIX Workshop on Offensive Technologies* (2007).

[44] WATSON, R. N. M., ANDERSON, J., LAURIE, B., AND KENNAWAY, K. Capsicum: Practical capabilities for unix. In *Proceedings of USENIX Security Symposium* (2010).

[45] WEINHOLD, C., AND HÄRTIG, H. VPFS: building a virtual private file system with a small trusted computing base. In *Proceedings of European Conference on Computer Systems* (2008).

[46] WEINHOLD, C., AND HÄRTIG, H. jVPFS: adding robustness to a secure stacked file system with untrusted local storage components. In *Proceedings of USENIX Annual Technical Conference* (2011).

[47] YANG, J., AND SHIN, K. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of ACM Conference on Virtual Execution Environments* (2008).

[48] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY T., OKASAKA, S., NARULA, N., FULLAGAR, N., AND GOOGLE INC. Native Client: A sandbox for portable, untrusted x86 native code. In *Proceedings of IEEE Symposium on Security and Privacy* (2009).

[49] 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. *Communications of the ACM 53*, 1 (2010), 91–99.

[50] ZHANG, F., CHEN, J., CHEN, H., AND ZANG, B. CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of ACM Symposium on Operating Systems Principles* (2011).

[51] ZHANG, Y., JUELS, A., REITER, M. K., AND RISTENPART, T. Cross-VM side channels and their use to extract private keys. In *Proceedings of ACM Conference on Computer and Communications Security* (2012).

Static Analysis of Variability in System Software: The 90,000 `#ifdefs` Issue[*]

Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, Daniel Lohmann
{tartler, dietrich, sincero, wosch, lohmann}@cs.fau.de
FAU Erlangen-Nürnberg

Abstract

System software can be configured at compile time to tailor it with respect to a broad range of supported hardware architectures and application domains. The Linux v3.2 kernel, for instance, provides more than 12,000 configurable features, which control the configuration-dependent inclusion of 31,000 source files with 89,000 `#ifdef` blocks.

Tools for static analyses can greatly assist with ensuring the quality of code-bases of this size. Unfortunately, static configurability limits the success of automated software testing and bug hunting. For proper type checking, the tools need to be invoked on a concrete configuration, so programmers have to manually derive many configurations to ensure that the configuration-conditional parts of their code are checked. This tedious and error-prone process leaves many easy to find bugs undetected.

We propose an approach and tooling to systematically increase the *configuration coverage* (CC) in compile-time configurable system software. Our VAMPYR tool derives the required configurations and can be combined with existing static checkers to improve their results. With GCC as static checker, we thereby have found hundreds of issues in Linux v3.2, BUSYBOX, and L4/FIASCO, many of which went unnoticed for several years and have to be classified as serious bugs. Our resulting patches were accepted by the respective upstream developers.

1 Introduction

System software typically employs compile-time configuration as a means to tailor it with respect to a broad range of supported hardware architectures and application domains. A prominent example is Linux, which in v3.2 provides more than 12,000 configurable features (KCONFIG options) that control the inclusion of 31,000 source files and 89,000 `#ifdef` blocks when building the Linux kernel. The huge and growing amount of configurability in modern system software implies quite some challenges with respect to testing and maintenance.

Static configurability is mostly implemented by source-code transformations [16, 17]: The build system and textual preprocessors, such as the *C Preprocessor* (`CPP`), interpret configuration flags to (a) filter the set of compilation units and (b) transform their actual content before passing them to the compiler. Consider this example of a *variation point* implemented with `CPP` in Linux:

```
#ifdef CONFIG_NUMA
  Block1
#else
  Block2
#endif
```

For any given configuration, depending on the configuration switch `CONFIG_NUMA`, *either Block*$_1$ *or Block*$_2$ is passed to the compiler (or any other static checker that drops in as a compiler replacement). This means that the responsible maintainer has to derive at least two configurations to validate that each line of code does even *compile*. This is not trivial: `CONFIG_NUMA` and the containing translation unit are constrained by further rules and configuration switches in the make files (KBUILD) and the feature model (KCONFIG) that all have to be set to the right values.

It is not hard to imagine that doing this manually does not work in practice. Nevertheless, this is the state of the art: Point 8 from the *Linux Kernel patch submission checklist*[1] requires, that all submitted code

> has been carefully reviewed with respect to relevant KCONFIG combinations. This is very hard to get right with testing – brainpower pays off here.

Our approach **replaces brainpower by tools**:

```
$ git am bugfix.diff  # Apply patch
$ vampyr -C gcc --commit HEAD  # Examine
```

[*] This work was partly supprted by the German Research Council (DFG) under grant no. LO 1719/3-1

[1] cf. `Documentation/SubmitChecklist` in the source tree

VAMPYR maximizes the *configuration coverage* (CC) by automatically deriving a set of configurations that together cover all variation points (`#ifdef` blocks and configuration-conditional files) in all files modified by the patch. It then invokes the build-system for each configuration, in this case with GCC as static checker.

1.1 Problem: Configuration Coverage

Many papers (e.g., [2, 4, 7, 18]) have been published about applying static bug-finding approaches to Linux and other pieces of system software. In all cases the authors could find a significant number of bugs. It is remarkable that the issues of configuration-conditional code and CC is not mentioned at all in these papers – the authors do not even state which configuration(s) they have analyzed. This does not only raise strong issues with respect to scientific reproducibility,[2] but also potentially limits their success: We have to assume that only a single configuration and architecture was analyzed (as also reported by Palix and colleagues [15]) – so how many bugs were missed that could have been found with full coverage?

There also is a more practical side of CC: A Linux developer, for instance, who has modified a bunch of files for some maintaining task would probably want to make sure that every edited line of code does actually compile and has been tested before submitting a patch. However, how to derive a set of configurations that covers all configuration-conditional pieces?

Deriving a configuration that reliably selects a particular configuration-conditional part of the code is not trivial. The problem is that for proper type checking the configuration needs to be sound and complete, as only then all header include paths are available and all types are properly resolved. To derive such a configuration, not only the C source code, but also the build system, feature model, and architecture have to be examined.

In practice, this leads to the situation that only a single architecture and configuration is checked. In the case of Linux, this typically is Linux/`x86` and – in the best case – the predefined `allyesconfig` configuration, which is supposed to be a maximum configuration. However, current versions of Linux support more than twenty architectures and `allyesconfig` is by far not a full configuration: Depending on the architecture, it covers only 42–83 percent of all configuration-conditional parts of the code. The result is that – despite extensive code reviews and other quality measures performed by the community – Linux contains quite some code that does not even compile.

1.2 Our Contributions

Our variability-aware driver VAMPYR mitigates these problems. It maximizes the CC by automatically deriving a set of configurations. By just employing the *compiler* (GCC) as a static checker, we thereby already can find hundreds of issues in Linux, L4/FIASCO, and BUSYBOX. In particular, we claim the following contributions:

(a) We analyze the conceptual and technical issues of configuration-dependent bugs (Section 2) and quantify how many variation points of the Linux source base are missed by the current state of the art (Section 4).

(b) We present an approach and tool implementation to systematically increase the CC in compile-time configurable system software (Section 3). Our approach and the resulting VAMPYR tool provide an easy and noninvasive integration into existing build systems and combination with existing code checkers, such as CLANG, GCC, SPARSE or COCCINELLE [14].[3] Besides Linux, we also have applied our approach to the L4/FIASCO μ-kernel and the BUSYBOX coreutils generator for embedded systems.

(c) Our experimental studies with GCC 4.7 as a static checker have revealed hundreds of issues (Section 5). For Linux/`arm` VAMPYR increases the CC (compared to `allyesconfig`) from 59.9 to 84.4 percent, which results in 199 additionally reported configuration-conditional issues (compiler warnings and errors). 91 of these issues have to be classified as serious bugs. We proposed patches for seven bugs in Linux and one in L4/FIASCO and BUSYBOX. All patches got accepted and the responsible developers have confirmed the found bugs. Some bugs went unnoticed for up to six years – just because they do not show up in a standard configuration.

1.3 Previous Work

This paper builds on previous work, especially the open-sourced variability extractors for KCONFIG, KBUILD, and CPP we have presented in [3, 20]. Our work on *variability defects* in Linux [20] reveals bugs in `#ifdef` expressions or KCONFIG constraints, such as typos in the feature identifiers or presence conditions that are a tautology/contradiction, so that the `#ifdef` statement or the complete block can be removed. In essence [20] finds faulty `#ifdef` statements, but does not look *inside* the thereby constrained blocks of code. This is what VAMPYR does by maximizing the CC of existing static checkers, so in this paper we look for a very different kind of configurability-related bugs.

In a previous workshop paper [19], we have sketched the issue of CC, the coverage of `allyesconfig` (re-

[2] In their "Ten years later" paper, Palix and colleagues describe the enormous difficulties to figure out the Linux v2.4.1 configuration used by Chou *et al.* in [2] in order to reproduce the results. Eventually, they had to apply source-code statistics to figure out the configuration "that is closest to that of Chou *et al.*" [15].

[3] VAMPYR and all related tools presented this paper are available for download under GPLv3 at `http://vamos.cs.fau.de/trac/undertaker`.

stricted to Linux/`x86`), and the idea to improve the CC by employing multiple configurations. However, we did not find any bug; our results regarding CC later turned out to be *way* too optimistic: The generated configurations were not sound, as we did not consider the coarse-grained variability implemented by the build system. This led to the development of our variability extractor for KBUILD [3], which we have integrated into our VAMPYR tool for this work. Only thereby, VAMPYR has become a practically usable driver for static checkers that has already helped to identify hundreds of issues in Linux, L4/FIASCO, and BUSYBOX.

2 Configuration-Dependent Bugs

The goal of our approach is to increase the effectiveness of existing static analysis tools so that they reveal also *configuration-dependent bugs*. Those are bugs that manifest only in configuration-conditional parts of the code, such as `#ifdef` blocks or configuration-conditional source files.

We classify a bug as a configuration-dependent bug, iff there exists any configuration in which the bug is not observed. In the following, we present some examples of configuration-dependent bugs we have found by maximizing the CC of GCC with VAMPYR:

(1) Consider the following situation in the HAL for the ARM architecture in Linux. In the file `arch/arm/mach-bcmring/core.c`, the timer frequency depends on the configured derivate:

```
#if defined(CONFIG_ARCH_FPGA11107)
/* fpga cpu/bus are currently 30 times slower so
     scale frequency as well to slow down Linux's
     sense of time */
[...]
#define TIMER3_FREQUENCY_KHZ (tmrHw_HIGH_FREQUENCY_HZ
     /1000 * 30)
#else
[...]
#define TIMER3_FREQUENCY_KHZ (tmrHw_HIGH_FREQUENCY_HZ
     /1000)
#endif
```

The variable `tmrHw_HIGH_FREQUENCY_MHZ` is defined in the header file `tmrHw_reg.h` with the value 150,000,000 to denote a frequency of 150 MHz. These timer frequencies are used in the static C99 initialization of the timer `sp804_timer3_clk`:

```
static struct clk sp804_timer3_clk = {
        .name = "sp804-timer-3",
        .type = CLK_TYPE_PRIMARY,
        .mode = CLK_MODE_XTAL,
        .rate_hz = TIMER3_FREQUENCY_KHZ * 1000,
};
```

The problem is that the member `rate_hz`, which has the type `unsigned long` (i.e., 32 bits on this platform), is too small to contain the resulting value of 30 times 150 Mhz in Hertz. We have reported this issue (unnoticed for three years and detected by our tool) to the responsible maintainer, who promptly acknowledged it as a new bug.[4] The point, however, is: This is a configuration-conditional bug that is *easy to detect* at compile time! The GCC compiler correctly reports it (with an integer overflow warning) iff (a) Linux is compiled for a 32-bit platform *and* (b) the Linux configuration happens to include the `#ifdef` block, which, however, is inserted by the CPP only if the `CONFIG_ARCH_FPGA11107` feature flag is set – which in turn depends on several other features.

(2) BUSYBOX is a compile-time tailorable implementation of the UNIX core utilities for memory-constrained environments. It is employed in many wireless routers and DSL modems, but also in several Linux distributions. With VAMPYR, we found several configurations of BUSYBOX, for which the GCC compiler warns about format-string security problems in `coreutils/stat.c`. The upstream developers have confirmed this issue as a new security-relevant bug.[5]

(3) In the L4/FIASCO μ-kernel, the file `ux/main-ux.cpp` revealed a compilation error, as the instantiation of a `Spin_lock` type lacks a type parameter. Again, this is a configuration-dependent bug, which is reported by GCC iff the feature flags `CONFIG_UX` (for choosing Linux user-mode as target architecture) *and* `CONFIG_MP` (for multi-processor support) are both enabled. We have reported this issue (detected by our tool) to the L4/FIASCO developers, who confirmed it as a new bug.

Summary All of the above bugs were caused by subtle issues that are difficult to spot with code reviewing, but easy to detect by a static checker (even the compiler in our case) – if the respective lines of code *are* getting compiled. However, the problematic lines of code are not covered by a standard configuration, which probably is the reason these bugs went unnoticed for up to three years. By increasing the CC, we have found tens of such issues in BUSYBOX and L4/FIASCO and hundreds in Linux (see Section 5).

3 Our Approach

The goal of our approach is, ultimately, to find configuration-conditional bugs by the systematic and automatic increasing of the *configuration coverage* (CC) of static code checkers and other quality measures, such as unit tests. Technically, this is achieved by automatically deriving a (reasonably small) set of configurations that together provide coverage of all configuration-conditional parts of the code. The static checker or unit test driver is then invoked individually for each element of this set.

[4] `https://lkml.org/lkml/2012/4/23/229`

[5] `http://lists.busybox.net/pipermail/busybox/2012-September/078360.html`

Hence, existing bug-hunting tools become *configurability-aware* without changing them. For now, we aim for *statement coverage*, that is, we want to make sure that every line of code is checked at least once by the employed static checker. We consider statement coverage as a significant first step to increase CC with modest computational complexity. While algorithms for higher coverage criteria are technically easy to integrate with our approach, we discuss their feasibility in Section 6.2.

The key idea is to extract configuration constraints and use a SAT Checker to construct sets of configurations. The challenge is that static variability is not only implemented by means of the `CPP`, but by a multitude of languages and tools that introduce and constrain variation points at different stages of the build process. In the following, we illustrate this on the example of the Linux generation process.

3.1 Static Variability in Linux

Linux is configured and generated in a sequence of steps that is depicted in Figure 1. Each of these steps effectively constraints the set of static variation points in the subsequent steps:

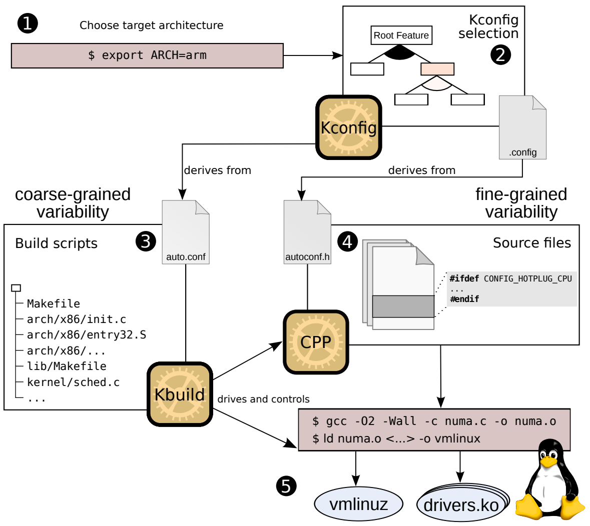
❶ The first decision is to choose a target architecture. Technically, this is done by setting an environment variable, which, if omitted, leads to native compilation; otherwise Kbuild uses a cross-compiler to produce a kernel for a nonnative architecture.

❷ Depending on the selected architecture, the Kconfig configuration tool loads a set of `Kconfig` files that together define the configuration space (features and constraints) of the chosen architecture. On Linux/`x86`, for instance, the user can choose from more than 7,700 features. Kconfig saves the resulting feature selection to a file (`.config`) that is used for storing, loading and interchanging feature selections. The generated artifacts (`.config` and `auto.conf`) control the compilation process in the subsequent steps.

❸ make is used to implement **coarse-grained variability** on a per-file basis. Depending on the configured features, the Kbuild tool selects the subset of all source files that are actually passed to the compiler and linker.

❹ In this subset, the `CPP` representation is used to implement **fine-grained variability** on a sub-file basis. Depending on the configured features (`configure.h`), `#ifdef` blocks are included or excluded by the `CPP` from the token stream passed to the compiler.

❺ Finally, make is also used to derive, depending on the selected features, compiler options and binding units, and thus, to drive the compilation and linking process. The result is a bootable kernel image and the associated *loadable kernel modules* (LKMs) for the chosen architecture and Kconfig selection.



```
$ export ARCH=arm
```

```
Makefile
arch/x86/init.c
arch/x86/entry32.S
arch/x86/...
lib/Makefile
kernel/sched.c
...
```

```
#ifdef CONFIG_HOTPLUG_CPU
...
#endif
```

```
$ gcc -O2 -Wall -c numa.c -o numa.o
$ ld numa.o <...> -o vmlinux
```

Figure 1: Fine-Grained and Coarse-Grained Variability Implementation in Linux.

Variation points are not only specified on the `CPP` level, but on different levels and in different languages. In fact, each build step (❶–❹) in Figure 1 also constitutes a distinct level of variability implementation, which constrains the effective number of variation points on subsequent levels: The chosen architecture ❶ constrains the possible Kconfig selection ❷, which in turn constrains the inclusion and exclusion of complete source files (coarse-grained variability) ❸, which further constrains the inclusion and exclusion of `#ifdef` blocks (fine-grained variability) ❹. To derive a concrete configuration that selects a particular `#ifdef` block in some translation unit, the developer basically has to go back all steps of this hierarchy to make sure that all dependencies of the translation unit and block are fulfilled.

The general lesson to be learned is that CC cannot be achieved by looking at the source code alone – all levels of static variability, including the constraints specified in the build system (Kbuild), feature model (Kconfig) and architecture selection have to be taken into account. This makes it so challenging for developers to manually derive configurations that cover all parts of their code.

3.2 Maximizing Configuration Coverage

The goal of the approach is to find a set of configurations for each source file that, when accumulated, selects all configuration-conditional parts of the code. Analyzing all configurations then maximizes the CC with respect to statement coverage.

Configuration-conditional parts of the code are given as complete files (level ❸, coarse-grained variability) and `#ifdef` blocks (level ❹, fine-grained variability). The resulting set of configurations has to cover both, but we conceptually operate on the most fine-grained level only: variation points that represent the selection (or deselection) of `#ifdef` blocks. Conditionally compiled files (level ❸) are treated as a single `#ifdef` block that (conceptually) spans the whole file content. Without loss of generality, in the following we therefore use *block* or *`#ifdef` block* as a collective noun for any kind of configuration-conditional variation point.

The reason why a single configuration is not the solution to this problem arises from the fact that blocks and whole source files may be in conflict to each other and can therefore not be enabled by the same configuration. Such conflicts can stem from *all* variability levels ❶–❹ in Figure 1, including the configuration model (KCONFIG) and architecture.

The CPP-statements of a C program describe a metaprogram that is executed by the C Preprocessor before the actual compilation by the C compiler takes place. In this meta-program, the CPP expressions (such as `#ifdef`– `#else`– `#endif`) correspond to the conditions in the edges of a loop-free[6] *control flow graph* (CFG); the thereby controlled `#ifdef` blocks are the statement nodes. On this CFG, established metrics, such as *statement coverage* or *path coverage*, can be applied.

The structure of CPP blocks and the identifiers used in their expressions translate into a propositional formula such that each CPP identifier and each `#ifdef` block is represented as a propositional variable. For the sake of a uniform treatment of source files with CPP blocks, we introduce an artificial CPP expression for the top-level block to express the constraints that are imposed by the build-system. For calculating the configurations, the actual block contents, in this case C code, can be ignored. We calculate the *Presence Condition* ($\mathcal{PC}$) for each `#ifdef` block, which here is influenced by three factors (Figure 2): Firstly, by φ_{CPP}, which encodes the structure and semantics of the CPP language (level ❹ variability). Secondly, by φ_{KBUILD}, the constraints that are imposed in build-system rules in KBUILD (level ❸ variability). Thirdly, by φ_{KCONFIG}, the constraints that arise from the feature dependencies declared in KCONFIG and by the selected architecture (level ❷ and ❶ variability).

The algorithm to calculate the configurations basically iterates over all blocks and employs a SAT solver to find a configuration that selects the current block. To reduce the number of SAT queries, blocks that are covered in already found configurations are skipped. As a further optimization, the algorithm tries to enable as many blocks as possible simultaneously, which reduces the number of resulting configurations. Our algorithm has a worst-case complexity of n^2 SAT calls for n blocks; however, in practice the number of SAT calls remains in the order of n for the vast majority of files. We discuss this algorithm in earlier work [19] with more detail.

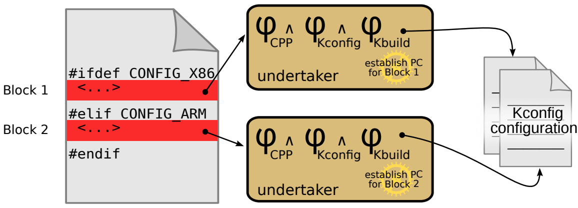

Block 1
Block 2
#ifdef CONFIG_X86
<...>
#elif CONFIG_ARM
<...>
#endif
$\varphi_{\text{CPP}} \wedge \varphi_{\text{Kconfig}} \wedge \varphi_{\text{Kbuild}}$
undertaker
establish PC for Block 1
$\varphi_{\text{CPP}} \wedge \varphi_{\text{Kconfig}} \wedge \varphi_{\text{Kbuild}}$
undertaker
establish PC for Block 2
Kconfig configurations

Figure 2: Deriving configurations: For each configuration-conditional block, we establish their $\mathcal{PC}$ to derive a set of configurations that maximize the CC.

3.3 Implementation: The VAMPYR

We provide the VAMPYR tool as an easy-to-use variability-aware driver that orchestrates the concepts and tools outlined in the previous sections. The general interaction between the individual tools is depicted in Figure 3. First, VAMPYR ensures that all variability constraints from CPP, KBUILD and KCONFIG are available. This formula is loaded (UNDERTAKER in Figure 3) and used to produce the configurations, on which the tools for static analysis are applied.

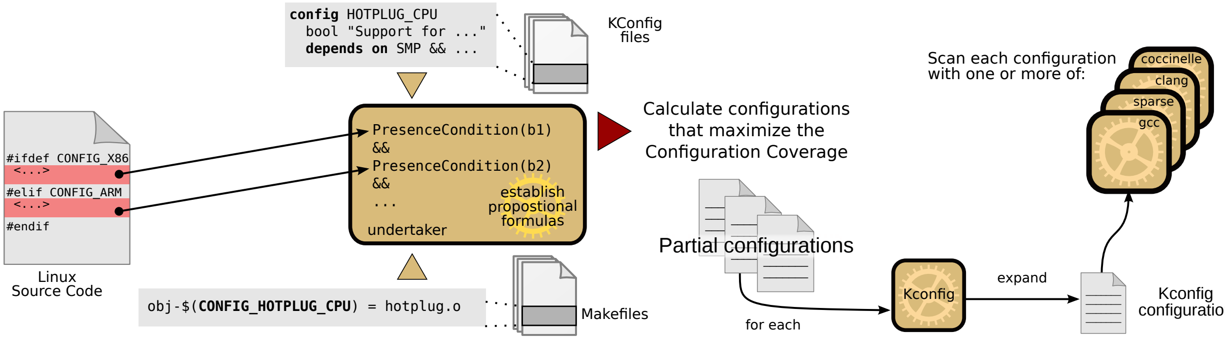
Note that the resulting configurations only cover variation points that are included in the source file, which means that they cannot be loaded directly into the KCONFIG configuration tool. Conceptually, we can understand this *partial configuration* as a set of variation points on level ❹ in Figure 1, for which we need to find a sound configuration on level ❷. This means that a produced configuration does not constrain the selection of the remaining thousands of configuration options that need to be set in order to establish a full KCONFIG configuration file that can be shared among developers. These remaining unspecified configuration options can be set to any value as long as they do not conflict with the constraints imposed by the partial configuration.

To derive configurations that can be loaded by KCONFIG, we reuse the KCONFIG tool itself to set the remaining unconstrained configuration options to values that are not in conflict with φ_{KCONFIG}. With these full configurations, we use the KBUILD build system to apply the tools for static analysis on the examined file. So far, we have integrated three different tools for static analysis: GCC, SPARSE, and SPATCH from the COCCINELLE tool-suite [14, 15].

3.4 Application Scenarios

(a) The Linux maintainer for the `bcmring` ARM development board has received a contributed patch via email. She first applies the patch to her local git tree, and

[6] Leaving aside "insane" CPP meta-programming techniques based on recursive `#include`, which are not used within Linux.



```
config HOTPLUG_CPU
  bool "Support for ..."
  depends on SMP && ...
```

```
#ifdef CONFIG_X86
  <...>
#elif CONFIG_ARM
  <...>
#endif
```

```
PresenceCondition(b1)
&&
PresenceCondition(b2)
&&
...
```

```
obj-$(CONFIG_HOTPLUG_CPU) = hotplug.o
```

KConfig files
Linux Source Code
establish propostional formulas
undertaker
Makefiles
Calculate configurations that maximize the Configuration Coverage
Partial configurations
for each
Kconfig
expand
Kconfig configuration
Scan each configuration with one or more of:
coccinelle
clang
sparse
gcc

Figure 3: Workflow of the VAMPYR configuration-aware static-analysis tool driver

then runs the VAMPYR tool on all files that the proposed change modifies:

```
$ git am bugfix.diff            # Integrate the patch
    as new commit
$ vampyr -C gcc --commit HEAD   # Examine files of
    latest commit
```

VAMPYR derives a CC-maximizing set for each file modified by the patch. The resulting configurations are plain text files in a syntax that is familiar to Linux developers, but only cover those variation points that are actually related to the $\mathcal{PC}$s contained in the affected source files. VAMPYR utilizes the KCONFIG configuration tool to set all remaining unspecified items to the default values. This expanded configuration is activated in the source tree (i.e., KBUILD updates the force-included `autoconf.h` and `auto.make` files), and GCC, or another static checker, is called on the examined file.

The issued warnings and errors are collected and presented to the developer after VAMPYR has inspected all configurations. The whole process takes less than a minute to complete on a modest quadcore development machine. In this case, VAMPYR reveals the integer overflow bug that has been presented in Section 2.

(b) The same maintainer implements a nightly quality-assurance run. After having integrated the submissions of various contributors, she calls it a day and lets VAMPYR check the complete source code base on Linux/`arm` (a worklist with 11,593 translation units) in a cronjob:

```
$ vampyr -C gcc -b worklist
```

In this case, VAMPYR calculates 14,222 configurations (~1.2 per file) in less than 4 minutes. The actual analysis, which includes extracting the variability from KCONFIG and KBUILD (cf. Section 3.1 and 3.2) and running the compiler and Linux makefiles, takes about 4.5 hours. We present and discuss the summarized findings of such a run in the evaluation (Section 5, Table 2 and 3).

Both application examples show that the approach results in a straight-forward and easy to use tool that unburdens developers from the tedious task of finding (and testing) the relevant configurations manually.

4 Configuration Coverage

As also reported by Palix and colleagues [15], we assume that static checkers and bug-finding tools are generally applied to a single configuration only. This raises the question of how many conditional blocks are commonly left uncovered. To be able to answer this question (and eventually quantify in Section 5 how much VAMPYR can improve on the situation), we first establish in Section 4.1 a metric for the effective CC achieved by a configuration or a set of configurations. We then use this metric in Section 4.2 to calculate the CC of the `allyesconfig` standard configuration in Linux. This synthetic configuration enables as many features as possible and is supposed to cover the maximum amount of code. We use it here to get an *upper bound* of the CC that can be achieved by testing a single configuration only.

4.1 Calculating Configuration Coverage

The definition of the CC depends on the chosen coverage criteria, including *statement coverage* (every block is included at least once), *decision coverage* (every block is included at least once and excluded at least once), and *path coverage* (every possible combination of blocks is included at least once). In this work, we go for *statement coverage* and define the CC of a given configuration as the fraction of the thereby *selected blocks* divided by the number of *available blocks*:

$$CC_S := \frac{\text{selected blocks}}{\text{available blocks}} \tag{1}$$

To calculate CC_S, we need to determine the number of *selected blocks* and the number of *available blocks* given by a configuration. To determine the set of selected blocks, we calculate the $\mathcal{PC}$ for each block b (as described in Section 3.2) to check if the respective block gets enabled by the configuration. For unconditional parts of the code (such as the file `fork.c`, which is included in every Linux configuration), the $\mathcal{PC}$ is a tautology.

Determining the set of available blocks is more complex: Depending on (a) the taken perspective (such as a concrete architecture) and (b) the constraints specified on each configuration level (❶–❹), many blocks are only *seemingly* configurable. To illustrate this effect, consider the following excerpt from `drivers/net/ethernet/broadcom/tg3.c`:

```
static u32 __devinit tg3_calc_dma_bndry(struct tg3 *
     tp, u32 val)
{
        int goal;
[...]
#if defined(CONFIG_PPC64) || defined(CONFIG_IA64) ||
     defined(CONFIG_PARISC)
        goal = BOUNDARY_MULTI_CACHELINE;
#else
[...]
#endif
```

This code configures architecture-dependent parts of the Broadcom TG3 network device driver with `#ifdef` blocks. Given any concrete architecture (which is the common perspective in Linux development), these blocks are not variable: The $\mathcal{PC}$s of the `#ifdef` blocks will either result in a tautology (on Linux/IA64, respectively Linux/PARISC) or a contradiction (on all other architectures) for any KCONFIG selection.

So, on level ❹ for Linux/`ia64` and Linux/`parisc` the `#if` part is always selected, whereas for all other architectures the `#else` part is chosen – but this only holds under the assumption, that on level ❸ the `tg3.c` file itself is selected. On Linux/`s390`, for instance, this file is singled out by a MAKE-file constraint; hence, the $\mathcal{PC}$ of *each* block is a contradiction. In line with our terminology from [20], we call a block with a $\mathcal{PC}$ that is a contradiction on the taken perspective, a *dead* block; a block with a $\mathcal{PC}$ that is a tautology is called an *undead* block, respectively. Both play an important role with respect to CC: A dead block cannot be selected by any configuration, whereas an undead block is implicitly selected by every configuration.

The Linux/`s390` example shows that it generally is not obvious from the code if a block is dead/undead: Most of the 12,000 Linux features (`CONFIG_` flags) become a tautology or contradiction because of the constraints expressed on level ❷ or ❸. On Linux/`x86`, for instance, 17 percent of all blocks are only seemingly variable, whereas on Linux/`s390`, this holds for 67 percent. One reason for this high rate is that `s390` hardware does not feature the PCI family of buses, so all PCI-related features, including many device drivers, become contradictions.

The practical consequence is that dead and undead blocks have to be singled out for calculating the configuration coverage. We call this refined metric the *normalized configuration coverage* (CC_N):

$$CC_N := \frac{\text{selected blocks} - \text{undead blocks}}{\text{all blocks} - \text{undead blocks} - \text{dead blocks}} \tag{2}$$

In Table 1, the CC_N is normalized with respect to the selected architecture in Linux, which impacts what blocks have to be considered as dead or undead. For a platform maintainer, for instance, ignoring blocks of a "foreign" architecture makes perfect sense: Linux is generally compiled natively and code parts for other architectures are likely to not compile anyway.

Architecture	Total kLOC	in CPP blocks	# variation points (dead/undead rate)	allyes $\mathcal{CC}_S$	allyes $\mathcal{CC}_N$
x86	8,391	4.5%	30,368 (17%)	65.2%	78.6%
hardware	6,417	3.6%	22,152 (22%)	59.7%	76.8%
software	1,974	7.6%	8,216 (4%)	80.2%	82.7%
arm	8,568	4.6%	33,356 (18%)	49.2%	59.9%
hardware	6,629	3.9%	25,140 (20%)	40.8%	51.2%
software	1,938	6.9%	8,216 (11%)	74.8%	83.6%
mips	7,848	4.3%	30,094 (23%)	42.3%	54.5%
hardware	5,896	3.3%	21,878 (29%)	30.1%	42.1%
software	1,952	7.4%	8,216 (7%)	74.8%	79.8%
s390	2,783	5.7%	28,756 (67%)	24.2%	72.1%
hardware	1,034	2.8%	20,540 (86%)	5.1%	37.2%
software	1,748	7.3%	8,216 (18%)	71.8%	86.8%
...			*< 19 further architectures >*		
Mean μ	6,447	3.8%	29,180 (39%)	43.9%	71.9%
Std. Dev. σ	±1,652		±1,159	±11.8%	±12.2%

Table 1: Quantification over variation points across selected architectures in Linux v3.2 and the corresponding $\mathcal{CC}_S$ and $\mathcal{CC}_N$ of `allyesconfig`.

4.2 The Configuration Coverage of 'allyesconfig' in Linux

Table 1 lists configurability-related source code metrics together with the resulting CC_S and CC_N of the `allyesconfig` standard configuration. We have examined these metrics for 24 out of the 27 Linux architectures. Table 1 lists an excerpt of this analysis: selected "typical" architectures (for PCs, embedded systems, mainframes), together with the mean μ and standard deviation σ over all 24 analyzed architectures.[7] We further discriminate the numbers between "hardware related" (originated from the subdirectories `drivers`, `arch`, and `sound`) and "software related" (all others, especially `kernel`, `mm`, `net`).

The average Linux architecture consists of 6,447 kLOC distributed over 8,231 source files, with 3.8% of all code lines in (real) `#ifdef` or `#else` blocks and 29,180 total variation points (`#ifdef` blocks, `#else` blocks, and configuration-dependent files). There is relatively little variance between Linux architectures with respect to these

[7] We could not retrieve results for `um`, `c6x`, and `tile`, which seem to be fundamentally broken. The complete data tables are available as an online appendix: `http://vamos.cs.fau.de/usenix2014-annex.pdf`

simple source-code metrics, as all architectures share a large amount of the source base (including the kernel and device drivers).

For the three rightmost columns, however, the variance is remarkable. The rate of dead/undead variation points varies from 17 percent on Linux/`x86` to up to 67 percent on Linux/`s390` (μ = 39%). It furthermore is reciprocally correlated to the CC_S of `allyesconfig`: These numbers underline the necessity to normalize the CC_S with respect to the actual viewpoint (here: the architecture). The normalized configuration coverage CC_N is generally much higher ($\mu = 71.9$, $\sigma = 12.2$) – here Linux/`s390` (72.1%) is even above the average and close to Linux/`x86` (78.6%).

The situation is different on typical embedded platforms, where the CC_N of `allyesconfig` is significantly lower: On Linux/`arm`, the largest and most quickly growing platform, only 59.9 percent are covered. These numbers are especially influenced by the relatively low CC_N (51.2%) achieved in the hardware-related parts. We find a similar situation for Linux/`mips`, for which only 54.5 percent are covered. We take this as an indicator for the larger hardware variability typically found on embedded platforms, which manifest in many alternative or conflicting features on the KCONFIG level and in `#else` blocks on the `CPP` level.

5 Evaluation

In the following, we evaluate the benefit of increasing the CC_N with VAMPYR. The working hypothesis is that especially subsystems and architectures with a relatively low CC_N of `allyesconfig` are prone to bugs that in principle are easy to find (reported by the compiler), but remain undetected for several years – like the integer overflow issue from Section 2.

We analyze this hypothesis (i.e., how many additional bugs can be found with our approach) on two operating systems, namely Linux version v3.2 and L4/FIASCO, as well as on BUSYBOX, a versatile user-space implementation of important system level utilities targeted at embedded systems. They all use sufficiently similar versions of KCONFIG, which allows reusing the variability extractor for φ_{KCONFIG} for all projects.

In all cases, we calculate a set of configurations for each file as described in Section 3.3, and apply GCC 4.7 as static checker for each configuration on all files individually. As an optimization, the initial starting set contains the standard configuration `allyesconfig`.

5.1 Application on Linux

On Linux, we use VAMPYR to generate the configurations for the 24 architectures examined in Section 4.2. In comparison to `allyesconfig`, VAMPYR increases the CC_N for every architecture, on average from $\mu = 71.5$ to $\mu = 84.6$ percent. Our Quad-core workstation calculates all partial configurations for 9,300 source files in less than 4 minutes. For the sake of comprehensibility, we limit in the following the in-depth analysis with GCC as static checker to three architectures. We choose, based on the observations in Section 4.2, `x86`, `arm`, and `mips`: We assume Linux/`x86` to be the best tested architecture – because of its maturity, wide-spread application, and the fact that it has the lowest rate of dead/undead blocks (see Table 1). Hence, we expect to find relatively fewer configuration-dependent bugs than in Linux/`arm`, which is the largest (in terms of files and code lines) and, driven by Android, most quickly growing Linux architecture, with a relatively low CC_N. We analyze Linux/`mips` as another embedded platform that is less in a state of flux than Linux/`arm`, but shows an even lower CC_N.

Table 2 depicts the results: Again we list the CC_N and found issues for both the hardware- and software-related parts of Linux. For the three architectures, we notice an increase of the CC_N by 10 to 36 percent, paid by about 20 percent more GCC invocations compared to a regular compilation with a single configuration. So on average, a Linux translation unit requires 1.2 invocations of a static checker to achieve CC with respect to statement coverage.

However, even though VAMPYR does increase the CC_N, it is not increased to 100 percent. We achieve the best result for Linux/`mips` with 91 percent coverage (`allyesconfig`: 55%); for the other two architectures the results are slightly lower. This is caused by (a) deficiencies in the current VAMPYR implementation as well as (b) bugs in the Linux KCONFIG models. We further discuss these issues in Section 6.1.

Nevertheless, VAMPYR reveals a high number of additional GCC messages that are not found with the `allyesconfig` configuration (last column of Table 2): 26 additional messages on Linux/`x86`, 199 on Linux/`arm`, and 91 on Linux/`mips`.

We take the number of `#ifdef` blocks per reported issue (bpi) as a normalization metric for code quality. This confirms our working hypothesis from Section 5: The bpi of Linux/`x86` is 110 , which is the lowest among the examined architectures. It is about 2.4 times better than the bpi of 46 revealed for Linux/`arm`, which is the highest. Linux/`mips` is with an bpi of 85 in between. On all architectures, the hardware-related parts of the source code (`arch`, `drivers`, `sound`) contain significantly more issues than the software-related parts (everything else, especially `kernel`, `mm`, and `net`) – we can confirm the frequent observation that hardware-related code contains significantly more bugs than software-related code [2, 15] also in the context of variability. This holds in particular for the quickly growing `arm` architecture, where the hardware-related parts show 5.6 times more issues than

Software Project	allyesconf $\mathcal{CC}_N$	VAMPYR $\mathcal{CC}_N$	Overhead: increase of GCC Invocations	GCC #warnings VAMPYR (allyesconfig)	GCC #errors VAMPYR (allyesconfig)	Σ Issues	#ifdef blocks per reported issue (bpi)	**Result**: increase of GCC messages
Linux/x86	78.6%	88.4%	21.5%	201 (176)	1 (0)	202	110	26 (+15%)
hardware	76.8%	86.5%	21.0%	180 (155)	1 (0)	181	82	26 (+17%)
software	82.7%	92.4%	22.7%	21 (21)	0 (0)	21	351	0 (+0%)
Linux/arm	59.9%	84.4%	22.7%	417 (294)	92 (15)	508	46	199 (+64%)
hardware	51.2%	80.1%	23.7%	380 (262)	92 (15)	471	34	194 (+70%)
software	83.6%	96.3%	19.5%	37 (32)	0 (0)	37	192	5 (+16%)
Linux/mips	54.5%	90.9%	22.0%	220 (157)	29 (1)	249	85	91 (+58%)
hardware	42.1%	88.2%	21.5%	174 (121)	17 (1)	191	72	69 (+57%)
software	79.8%	96.3%	23.2%	46 (36)	12 (0)	58	128	22 (+61%)
L4/FIASCO	99.1%	99.8%	see text	20 (5)	1 (0)	21	see text	16 (+320%)
Busybox	74.2%	97.3%	60.3%	44 (35)	0 (0)	44	72	9 (+26%)

Table 2: Results of our VAMPYR tool with GCC 4.7 (`-fno-inline-functions-called-once -Wno-unused`) as static checker on Linux v3.2, L4/FIASCO and BUSYBOX.

the software-related parts.

In Table 2 the reported issues are differentiated between errors and warnings. However, this classification by the *compiler* is only seemingly related to the severity of the issue. While many developers consider warnings as more-or-less cosmetic issues, they often point to critical bugs, such as the integer overflow from Section 2. Nevertheless, by the fact that a high number of warnings is also revealed by allyesconfig, we have to conclude that at least some warnings are considered as “false positives”.

To quantify the actual benefit of our approach in this respect, we have reviewed all messages on Linux/arm manually to discriminate less critical messages from real bugs. The results of a conservative classification[8] are depicted in Table 3: 91 out of the 508 reported issues for Linux/arm have to be considered as real bugs. For seven bugs, including the issues from Section 2, we have proposed a patch[9] to the upstream developers, which all got immediately confirmed or accepted. Six of these seven bugs had been unnoticed for several years.

Less critical GCC messages	warnings	errors
Σ Less critical messages	347 (223)	16 (0)
Manually validated bugs		
Undeclared types/identifiers		46 (4)
Access to possibly uninitialized data	22 (20)	
Out of bounds array accesses	11 (7)	2 (0)
Bad pointer casts	8 (0)	
Format string warnings	1 (0)	
Integer overflows	1 (0)	
Σ Bugs found	**43 (27)**	**48 (4)**
Σ All reported issues	390 (250)	64 (4)

Table 3: Classification of GCC warnings and errors revealed by the VAMPYR tool on Linux/arm. The numbers in parentheses indicate messages that are also found when compiling the configuration allyesconfig

5.2 Application on L4/Fiasco

In order to show the general applicability, we apply the VAMPYR tool also to the code base of the L4/FIASCO μ-kernel. Compared to Linux, L4/FIASCO is relatively small: It encompasses about 112 kLOC in 755 files (only counting the core kernel, that is, without user-space packages). Nevertheless, we identify 1,255 variation points (1,228 conditional code blocks and 16 conditionally compiled source files) in the code base.

L4/FIASCO employs the KCONFIG infrastructure to configure 157 features on 4 architectures. Unlike Linux, the architectures are user-selectable KCONFIG options. Also, L4/FIASCO does not only use the CPP, but also uses a transformation process that allows programmers to declare interface and implementation in the same source file. This additional processing step produces traditional header and implementation files, which are preprocessed by CPP and compiled with GCC. We cope with this by processing the resulting CPP #ifdef blocks for calculating the configurations. However, because of the additional preprocessing step, the metrics of GCC invocations per source file and bpi do not relate to the results of Linux and BUSYBOX, so we leave them out in Table 2.

For L4/FIASCO, the VAMPYR tool produces 9 different configurations that in total cover 1,228 out of 1,239 #ifdef blocks, which maps to a CC_N of 99.8 percent. Compared to allyesconfig, the number of compiler messages thereby increased from 5 to 21, among them

[8] Only messages for which the manual source-code review provides *strong* evidence of an actual bug are counted as such. Everything else is considered to be less critical. We also count some errors (caused by #error statements) that point to issues in the KCONFIG model and not in the code as less critical.

[9] http://vamos.cs.fau.de/usenix2014-annex.pdf

the compilation error in `ux/main-ux.cpp` we have illustrated in Section 2. We have reported this issue to the L4/FIASCO developers, who confirmed it as a bug.

5.3 Application on Busybox

Another popular software project that makes use of KCONFIG is the BUSYBOX tool suite. The analyzed version `1.20.1` exposes 879 features that allow users to select exactly the amount of functionality that is necessary for a given use case, implemented by 3,316 `#ifdef` blocks and conditionally compiled source files.

For BUSYBOX, VAMPYR increases the number of reported issues from 35 to 44; we have proposed a fix for one of them to the upstream developers, who have confirmed it as a bug and accepted our patch.

6 Discussion

6.1 Threats to Validity – Quality of Results

Is it fair to compare to `allyesconfig`? In this paper we argue that checking a single configuration is not enough and evaluate this by comparing our configuration-aware VAMPYR results against the `allyesconfig` standard configuration, which we assume to achieve the best possible CC for a single configuration.

However, this is not guaranteed, as `allyesconfig` is a synthetic configuration generated by the KCONFIG tool with a simple algorithm: Traverse the feature tree and select each feature that is not in conflict to an already selected feature. The outcome is sensitive to the order of features in the feature tree, hence, does not necessarily include the possible maximum number of features. Also, even if we assume a maximum number of features as the outcome, this does not necessarily imply the largest possible CC, as we might have missed a feature with a highly crosscutting implementation (i.e., a feature that contributes many `#ifdef` blocks) in favor of another feature that contributes just a single variation point. However, features in Linux are generally not very cross cutting: 58 percent of all features in Linux v3.2 are implemented by a single variation point; only 9 percent contribute more than 10 variation points, most of which are architecture-related features that anyway cannot be modified on the KCONFIG level. So, despite these limitations, we consider `allyesconfig` as a realistic upper bound of the CC that could be achieved with a single configuration in practice.

Why not hundred percent coverage? Even though VAMPYR increases the CC significantly, we do not get full coverage. In some cases, the expansion of a partial configuration by KCONFIG results in a full configuration that contradicts the partial configuration. In order to achieve correct results (and in contrast to our in retrospect naïve attempt in [19]), VAMPYR validates the soundness of each configuration after the expansion process: Configurations that no longer contain the relevant features of interest are skipped; the thereby induced `#ifdef` blocks or files are considered as not covered in Table 2, which results in rates that are below hundred percent. We see three major causes for this effect:

(1) One reason for failed expansions are bugs in the Linux variability descriptions (KCONFIG models). Feature dependencies are notoriously hard to get right with KCONFIG, as the KCONFIG tool does not validate the soundness of the models: In feature dependency expressions, the KCONFIG language provides (via the `SELECT` statement) the option to select arbitrary other features; in this case it is in the responsibility of the *developer* to ensure that thereby the configuration remains valid. In practice, this is not always the case and leads to user-selectable configurations that are formally invalid (contain a contradicting feature), but nevertheless "work" for the user. However, as element of a partial configuration derived by VAMPYR, such a contradicting feature usually causes an incorrect expansion.

(2) Bugs in the KCONFIG descriptions can furthermore cause missing of some dead/undead blocks. This directly leads to a lower CC_N, as dead/undead blocks are substracted in denominator of the CC_N (see Equation 2).

(3) Another potential cause for expansion issues is that the VAMPYR implementation, in particular the model extractor for φ_{KCONFIG}, is not yet feature complete. We currently do not correctly handle the situation when some feature depends on the *value* (rather than the mere *selection*) of some other string or integer feature. Luckily, the number of features that employ value tests in their $\mathcal{PC}$ is low in Linux (`mips`:0.26%, `arm`:0.28% `x86`:0.31%; μ = 0.4%, σ = 0.22). Nevertheless, this can make the expansion fail, and thus, impact the achieved CC.

Are there false positives/negatives? The expansion issues imply that there is a high probability of false negatives – bugs we miss, because we do not achieve full CC. Nevertheless, our results show that our approach helps to discover a significant number of long-time overlooked bugs in Linux, L4/FIASCO, and BUSYBOX. The point is that VAMPYR is easy to use and does, by construction, not produce any false positives. Hence, the "annoyance factor" is low, which increases the chance of acceptance by system software developers.

6.2 Higher Coverage Criteria

The chosen coverage criterion also implies the existence of false negatives (i.e., undetected issues): The current implementation of VAMPYR achieves *statement coverage*, that is, every configuration-conditional block is included at least once – at the price of 20 percent additional compiler invocations. Would using a higher coverage criterion would reveal more issues?

We are currently experimenting with a VAMPYR prototype that provides *decision coverage*, that is, every block is included and excluded at least once. Technically, this is realized by virtually adding an empty `#else` block to every `#if` block without an `#else` part. Over the three analyzed architectures (Linux/`x86`, Linux/`arm`, Linux/`mips`), the shift from statement to decision coverage increases the number of reported issues by an additional nine percent at the price of fifteen percent more compiler invocations. We consider this as still acceptable for most use-cases.

The next step will be *path coverage*: every possible combination of blocks is included at least once. This has exponential overhead: a source file with n `CONFIG` flags requires up to 2^n configurations (compiler invocations) to achieve path coverage. However, we expect the constraints from the CPP, KBUILD, and KCONFIG levels to considerably restrict the number of actually possible configuration. Furthermore, more than 98 percent of all Linux compilation units employ five or less `CONFIG` flags.[10] So in practice, also path coverage may be feasible for large parts of Linux – at least for tasks such as checking a contributed patch (application scenario (a) from Section 3.4), which we consider as the major use case for VAMPYR.

Checking of a complete architecture with path coverage is probably infeasible on the average developer's machine. Heuristic approaches, such as *pairwise testing* [13], may be more promising candidates to achieve higher coverage at acceptable run times. This is a topic of further research. The point here is that we intend VAMPYR to be a tool that developers can use in their regular development cycles, so the run-time caused by algorithmic complexity (another "annoyance factor") has to be limited.

6.3 General Applicability of the Approach

We have evaluated our approach with Linux, L4/FIASCO, and BUSYBOX and GCC as a static checker. However, the approach is as well applicable to other software families and static checkers. To apply VAMPYR to some piece of configurable software, one basically needs extractors for the configuration points and constraints specified on all employed implementation levels of variability. These levels are generally project-specific: In Linux, we have the `ARCH` environment variable, KCONFIG, KBUILD, and CPP; in L4/FIASCO we additionally have the custom `preprocess` level. Our approach, however, makes it easy to integrate such custom variability extractors. We also expect a significant amount of reusability: Almost every system software project employs CPP to implement variability and the KCONFIG language is adopted by more and more projects as a means to describe the intended configurability.

Static checkers are not always warmly welcomed – many developers are (initially) reluctant to accept their findings [1]. To convincingly illustrate the issue of configuration-dependent bugs, we therefore have chosen GCC as our static checker: The compiler is a "least common denominator" of a bug-finding tool that *has* to be accepted by developers. Nevertheless, VAMPYR can be employed as variability-aware driver for any static checker that drops in as a compiler replacement. We have also tested it successfully with COCCINELLE [14, 15] and SPARSE. With SPARSE, for instance, VAMPYR more than doubles the issues reported for Linux/`arm`: from 9,484 (`allyesconfig`) to 23,964 (VAMPYR). The high number of issues already reported for `allyesconfig`, however, is a strong indicator that the "annoyance factor" of SPARSE is too high to be accepted as a helpful static checker by the Linux developers – even though "Check cleanly with sparse" is an explicit requirement on the Kernel patch submission checklist.

7 Related Work

Despite these apparent acceptance problems by kernel developers, automated bug detection by examining the source code has a long tradition in the systems community. Many approaches have been suggested to extract rules, invariants, specifications, or even misleading source-code comments from the source code or execution traces from Linux [2, 4, 7, 8, 11, 18]. However, it is remarkable that all of these papers seem agnostic to the used configuration and do not even mention what configuration has been analyzed – even though the wide-ranged analysis of feature implementations in system software by Liebig, Kästner, and Apel [9] underlines the impressive amount of CPP-based configurability in today's system software. The issue of CC is largely underestimated.

In the verification and validation community, the notion of CC has been defined in a very similar way to this work by Maximoff et al. [12] in the context of NASA spacecrafts. Unlike our work, this work does not create the configurations from the implementation. Instead, the CC assesses the quality of a given set of test cases.

Liebig et al. [10] present a good overview over the current state of the art for variability-aware analysis of software systems, in which the authors identify two major approaches to the problem: Either by making all tools for static analysis configurability-aware, or by improving the effectiveness of the existing tools by a configurability-aware driver by applying the tools using a sample set, that is, a subset of all possible configuration. Our VAMPYR tool is the latter; there are, however, several research groups that attempt the first approach: Kästner et al. [6] propose a technique coined *variability aware parsing*, which basically integrates the CPP variability into tools for static analysis and allows variability aware type-checking across all configurations. Gazzillo and Grimm

[10] The extreme corner-cases of the remaining two percent are `sysctl.c` with 59 flags and `sched.c` with 47 flags.

[5] propose a generalized and much better performing configurability-aware parser for C *with* CPP. Their SUPERC basically treats C and CPP together as a single language and thereby could be used as front-end for the implementation of variability-aware checkers. However, for any practical use, both approaches also need further assistance by a model of all variability constraints from other implementation levels, otherwise, type-checking invalid configurations would cause a very high run time and result in a tremendous number of false positives. The big advantage of these approaches is that they achieve path coverage over the CPP meta program. The disadvantages are that they only work on the CPP level and that they cannot be combined with other static checkers. Hence, we consider our VAMPYR approach to be more flexible.

8 Conclusions

System software is typically configured at compile-time to tailor it with respect to the supported application or hardware platform. Linux, for instance, provides in v3.2 more than 12,000 configurable features. This enormous variability imposes great challenges for software testing with respect to *configuration coverage* (CC).

Existing tools for static analyses are configurability-agnostic: Programmers have to manually derive concrete configurations to ensure CC. For this, they do not only have to consider the structure of #ifdef blocks in the code, but also the thousands of constraints specified in the build system and feature model. Hence, many easy-to-find bugs are missed, just because they happen to be not revealed by a standard configuration – Linux contains surprisingly much source code that does not even compile.

With our VAMPYR approach and implementation, the necessary configurations can be derived automatically. VAMPYR is easy to integrate into existing tool chains and provides configurability-awareness for arbitrary static checkers. With GCC as a static checker, we have revealed hundreds of configuration-dependent issues in Linux, L4/FIASCO, and BUSYBOX. For Linux/arm, we have found 60 new bugs, some of which went unnoticed for six years. We have also found one bug in L4/FIASCO and nine issues in BUSYBOX. For all three projects, the upstream developers have confirmed the reported bugs and accepted our resulting patches.

The tools VAMPYR and UNDERTAKER are available under GPLv3 at http://vamos.cs.fau.de/. All raw data is generated by automated experiments to support scientific reproducibility, and can be examined at http://vamos.cs.fau.de/jenkins (login: public/i4guest). The Linux data and a detailed description of our patches is furthermore available as an online-appendix at http://vamos.cs.fau.de/usenix2014-annex.pdf.

References

[1] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *CACM* 53 (2).

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. “An empirical study of operating systems errors”. In: *SOSP ’01*.

[3] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. “A Robust Approach for Variability Extraction from the Linux Build System”. In: *SPLC ’12*.

[4] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. “Bugs as deviant behavior: a general approach to inferring errors in systems code”. In: *SOSP ’01*.

[5] Paul Gazzillo and Robert Grimm. “SuperC: parsing all of C by taming the preprocessor”. In: *PLDI ’12*.

[6] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. “Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation”. In: *OOPSLA ’11*.

[7] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. “From uncertainty to belief: inferring the specification within”. In: *OSDI ’06*.

[8] Zhenmin Li and Yuanyuan Zhou. “PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code”. In: *ESEC/FSE ’00*.

[9] Jörg Liebig, Christian Kästner, and Sven Apel. “Analyzing the discipline of preprocessor annotations in 30 million lines of C code”. In: *AOSD ’11*.

[10] Jörg Liebig, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. “Scalable Analysis of Variable Software”. In: *ESEC/FSE ’13*.

[11] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. “Learning from mistakes: a comprehensive study on real world concurrency bug characteristics”. In: *ASPLOS ’08*.

[12] J.R. Maximoff, M.D. Trela, D.R. Kuhn, and R. Kacker. “A method for analyzing system state-space coverage within a t-wise testing framework”. In: *4th anual IEEE Systems Conference*.

[13] Sebastian Oster, Florian Markert, and Philipp Ritter. “Automated Incremental Pairwise Testing of Software Product Lines”. In: *SPLC ’10*. Vol. 6287.

[14] Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. “Documenting and Automating Collateral Evolutions in Linux Device Drivers”. In: *EuroSys ’08*.

[15] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. “Faults in Linux: Ten years later”. In: *ASPLOS ’11*.

[16] Henry Spencer and Gehoff Collyer. “#ifdef Considered Harmful, or Portability Experience With C News”. In: *USENEX ATC ’92*.

[17] Diomidis Spinellis. “A Tale of Four Kernels”. In: *ICSE ’08*.

[18] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. “/*iComment: Bugs or Bad Comments?*/”. In: *SOSP ’07*.

[19] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. “Configuration Coverage in the Analysis of Large-Scale System Software”. In: *PLOS ’11*. DOI: 10.1145/2039239.2039242.

[20] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. “Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem”. In: *EuroSys ’11*. DOI: 10.1145/1966445.1966451.

Yat: A Validation Framework for Persistent Memory Software

Philip Lantz Subramanya Dulloor Sanjay Kumar Rajesh Sankaran
Jeff Jackson

Intel Labs

Abstract

This paper describes the design and implementation of *Yat*. Yat is a hypervisor-based framework that supports testing of applications that use Persistent Memory (*PM*)—byte-addressable, non-volatile memory attached directly to the memory controller. PM has implications on both system architecture and software. The PM architecture extends the memory ordering model to add software-visible support for *durability* of stores to PM. By simulating the characteristics of PM, and integrating an application-specific checker in the framework, Yat enables validation, correctness testing, and debugging of PM software in the presence of power failures and crashes. We discuss the use of Yat in development and testing of the Persistent Memory File System (*PMFS*), describing the effectiveness of Yat in catching and debugging several hard-to-find bugs in PMFS.

1 Introduction

We are witnessing growing interest in Non-Volatile DIMMs (NVDIMMs) that attach storage class memory (e.g., *PCM*, *MRAM*, etc.) directly to the memory controller [5]. We refer to any such byte-addressable, non-volatile memory as Persistent Memory (*PM*).

PM has implications on system architecture and software [2]. Since PM performance—both latency and bandwidth—is within an order of magnitude of DRAM, software can map PM as write-back cacheable for performance reasons. Several studies [2, 6] have shown significant performance gains from the use of in-place data structures in write-back PM. These studies also show the need for extensions to the existing memory model to allow PM software to control *ordering* and *durability* of stores to write-back PM.

However, such extensions to the memory model introduce the possibility of new types of programming errors. For instance, consider a PM software flow as shown in Figure 1. Starting at consistent state A, PM software performs two writes to PM (set $W_{A->B}$), dirtying two cachelines (L_1, L_2) in the process. For traditional block based storage, software explicitly schedules IO to make these cachelines persistent at block granularity. However, for PM-based storage these cachelines can become persistent in arbitrary order by cacheline evictions outside of software control. Hence, extra care must be taken in enforcing ordering on updates to PM. Programmers today are not used to explicitly tracking and flushing modified cachelines in volatile memory. But, this is a critical requirement of PM software, failing which could cause serious consistency bugs and data corruption.

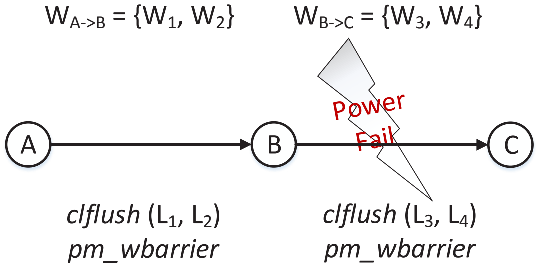


Figure 1: PM software flow

Testing for the correctness of PM software is challenging. One way is to simulate or induce failures (such as from power loss or crashes) and use an application-specific checker tool (similar to fsck for Linux filesystems) to verify consistency of the data in PM. However, this method tests only the actual ordering of writes to PM in a single flow, even though many other orderings are possible outside the control of the PM software (e.g., due to arbitrary cacheline evictions).

To overcome this challenge, we built *Yat* (meaning "trial by fire" in Sanskrit), a hypervisor-based framework for testing the correctness of PM. Yat uses a record and replay method to simulate architectural failure conditions that are specific to PM. We used Yat in validation and correctness testing of PMFS [2], which is a reasonably large and complex PM software module. Though we focus on PMFS as the only case study in this paper, the principles of Yat are applicable to other PM software, including libraries and applications [6].

Contributions of this paper are as follows:

- Design and implementation of Yat, a hypervisor-based framework for testing PM software.
- Evaluation of a Yat prototype, using PMFS as an example PM software.

2 System Architecture

We assume the Intel64 based system architecture described elsewhere [2], where software can access PM directly using regular CPU load and store instructions. Because PM is typically mapped write-back (for performance reasons), PM data in CPU caches can be evicted and made durable at any time. To give software the ability to control consistency, the architecture must provide a software visible guarantee of ordering and durability of stores to PM. The proposed architecture includes a simple hardware primitive, PM write barrier (*pm_wbarrier*), which guarantees durability of all stores to PM that have been explicitly flushed from the CPU caches but might still be be in a volatile buffer in the memory controller or in the PM module.

In Figure 1, to effect $W_{A->B}$ before $W_{B->C}$, PM software must flush the dirty cachelines L_{1-2} and make those writes durable using *pm_wbarrier*, before proceeding to writes in $W_{B->C}$. In complex software, it can be challenging to keep track of all the dirty cachelines that need to be flushed before the use of *pm_wbarrier*. We expect user-level libraries and programming models to hide most of this programming complexity from PM applications [6].

Yat is designed to help validate that PM software correctly uses cache flushes (*clflush*), ordering instructions (*sfence* and *mfence*), and the new hardware primitive (*pm_wbarrier*) to control durability and consistency in PM, even in the face of arbitrary failures and cacheline evictions. For any sequence of updates to PM, Yat creates all possible states in PM based storage and then runs the PM application's recovery tool to test recovery to a consistent state. The possible orderings are determined by the memory ordering model of the processor architecture. The proposed system (based on Intel64 architecture) follows these rules:

1. When a write is executed, it may become durable immediately or at any subsequent time up to the point where it is known to have become durable by rule 5.
2. When a write is executed that modifies a cache line that has been modified by a prior write, the later write is guaranteed to become durable no sooner than the prior write to the same cache line. This guarantee holds across cores based on Intel64 architecture.
3. When a *clflush* is executed, it has no effect until it is followed by a fence on the same processor.
4. When a fence is executed, prior clflushes on that processor take effect. All writes performed by any processor to the cache lines affected by the clflushes are flushed.
5. When a *pm_wbarrier* primitive is executed, all writes that have been flushed by rule 4 are made durable. Any writes that have not been clflushed—or that were clflushed but where no subsequent fence was executed on the same processor as the *clflush* prior to the *pm_wbarrier*—are not guaranteed to be durable.

3 Yat Design

Yat is a framework for testing PM software. We refer to the PM software being tested as *App*.

The goals of Yat are:

1) to test App for bugs caused by improper reordering of write operations; e.g., due to missing or misplaced ordering and durability instructions.

2) to exercise the PM recovery code in App in the context of a large variety of failure scenarios, such as power failures and software failures internal or external to App that cause it to abort.

To test that App applies sufficient memory ordering constraints to preserve consistency no matter when a failure occurs, Yat simulates reordering PM writes in every allowed order in which they may become durable in the PM hardware based on the rules in §2. Note that if App is multithreaded, Yat records the actual sequence of operations executed by the various threads. However, Yat does not model the non-determinism in the software as it has no knowledge of synchronization done by the software.

The operation of Yat is shown in Figure 2. Yat operates in two phases. The first phase, *Yat-record*, simply collects a trace (*App-trace*) while App is executing. Yat-record logs write and *clflush* instructions within the address range of PM, along with the explicit ordering instructions (sfence and mfence), and the new hardware primitive *pm_wbarrier*.

The second phase, *Yat-replay*, has the following steps:

Segment Yat-replay divides App-trace into segments, separated by each *pm_wbarrier* in the trace. For each segment, there is a set of writes to PM that have been executed but are not guaranteed to be durable. This set of writes is called the *active set* for that segment.

Reorder For each segment, Yat-replay selects subsets of writes in the active set that do not violate rule 2. Each such selected subset of writes is called a *combination*.

Replay For each combination, Yat-replay starts with an initial state (*App-initial-state*), applies all writes that are guaranteed to have become durable as of the *pm_wbarrier* that precedes the current segment, and then applies the writes contained in the current combination.

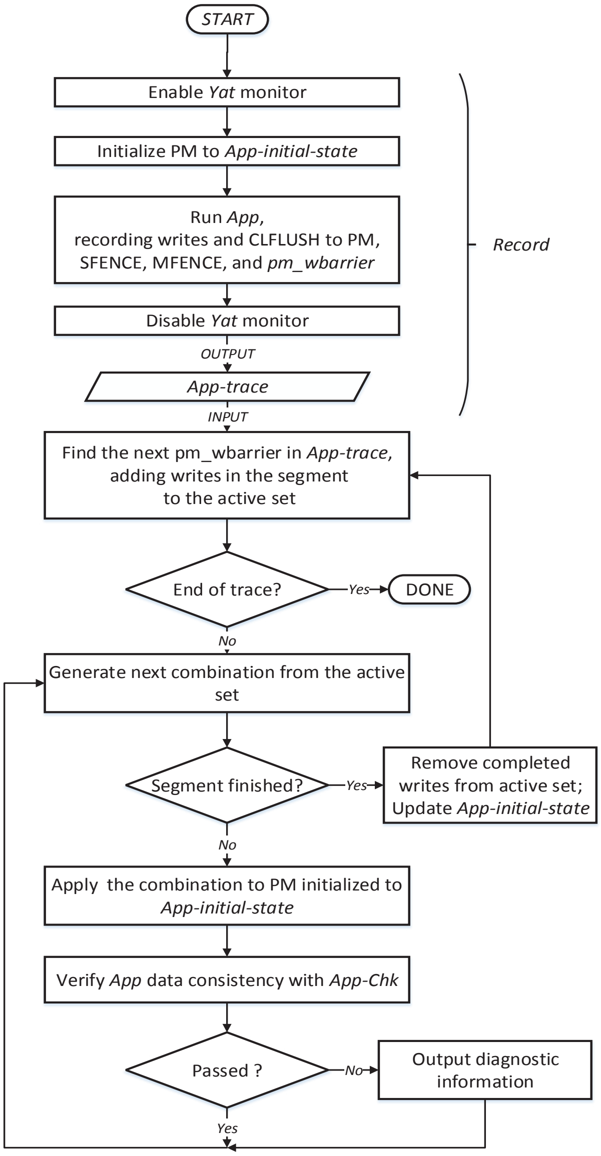

START
Enable Yat monitor
Initialize PM to App-initial-state
Run App,
recording writes and CLFLUSH to PM,
SFENCE, MFENCE, and pm_wbarrier
Disable Yat monitor
Record
OUTPUT
App-trace
INPUT
Find the next pm_wbarrier in App-trace,
adding writes in the segment
to the active set
End of trace?
Yes
DONE
No
Generate next combination from the active set
Segment finished?
Yes
Remove completed writes from active set;
Update App-initial-state
No
Apply the combination to PM initialized to App-initial-state
Verify App data consistency with App-Chk
Passed ?
No
Output diagnostic information
Yes

Figure 2: Yat flow diagram

Recover Yat-replay then runs App to restore the PM state from the point of the simulated failure to a consistent state. In the case of PMFS, this step consists of simply mounting the file system. For other applications, it may involve running an application-specific recovery routine.

Verify Yat-replay then runs an application-specific data integrity checker (*App-chk*), such as those used with file systems (fsck) and databases, to verify consistency of App's persistent data. If the check fails, Yat-replay reports the point in the trace where the failure occurred, along with the combination of uncommited writes that were applied.

The number of combinations grows exponentially with the number of writes in the active set, and could become prohibitively large. However, we found this issue to be less of a problem than one might fear. Because of rule 2, the number of cache lines modified by writes in the active set is more significant than the total number of writes. For instance, in PMFS, the number of cache lines written per segment is typically between four and six, so the number of combinations is typically manageable.

Before generating any combinations for a segment, Yat can compute the maximum number of combinations for the segment based on the number of write operations and affected cache lines in the active set. If this number is below a configurable threshold, Yat exhaustively tests every combination; otherwise, it chooses combinations at random. The decision on whether to do random or exhaustive testing is made separately for each segment of the trace. The threshold for combinations per segment must take into account both Yat performance and desired test coverage for App. For testing PMFS, we used a threshold of 250 combinations per segment.

4 Yat Implementation

Although the design of Yat is independent of the software being tested, some of the implementation choices were made with PMFS in mind. PMFS is a POSIX compliant file system, designed and optimized for PM and our system architecture. PMFS maintains both meta-data and data as a B-tree in PM and uses a combination of *atomic in-place updates* and *fine-grained logging* to ensure consistent updates to the meta-data. PMFS is a reasonably complex source base, with about 10K lines of code—ideal for a realistic evaluation of Yat.

Since PMFS runs in kernel mode, we needed a layer of software below the kernel to trap accesses to PM. Yat-record is implemented using our internal Type-I research hypervisor (similar to Xen [1]), called Hypersim. Like other VMMs, Hypersim uses *Intel64 VT-x* [4] to place itself between the hardware and the guest. Unlike a VMM, Hypersim does not actually virtualize the platform, but only provides a way to observe the behavior of a single guest. Hypersim uses typical VMM memory-management capabilities, such as Extended Page Tables [4], to cause VM exits on write accesses to PM, and then logs these events. Hypersim also intercepts *clflush* and fence instructions and *pm_wbarrier* hardware primitive and logs these events. Because these instructions normally do not cause VM exits, the current implementation requires App to be recompiled, substituting illegal instructions for the *clflush* and fence instructions and the *pm_wbarrier* hardware primitive. Hypersim intercepts these illegal instruction faults and logs them. This modification is not difficult in practice because often these instructions and primitives are implemented as macros or inline functions, requiring very few changes to App. Ideally, however, Yat-record would accomplish this without modifying App's source code. We intend to explore this capability in future work, perhaps using binary translation techniques [3].

Yat-replay is implemented jointly between a shell script and Hypersim. The shell script controls the steps of replay operation, while Hypersim manages App's PM state and maintains the current segment and combination being replayed.

When starting Yat-replay, the shell script loads App-initial-state and App-trace into Hypersim's memory. For each test, the shell script calls Hypersim to apply the next combination of writes to the PM. Hypersim applies the writes in the current combination using Copy-on-Write (CoW). The shell script then invokes App to perform recovery from the simulated failure, and App-chk to verify App's data consistency following the recovery.

Either App recovery code or App-chk may report a failure or may, in fact, hang, crash, or abort, depending on the nature of the problem. Yat-replay reports any of these types of failures, along with the segment of the trace that is being replayed, and the combination.

Since Hypersim uses CoW both for writing the combination and during recovery, it can easily restore PM to App-initial-state after each test, before applying the next combination.

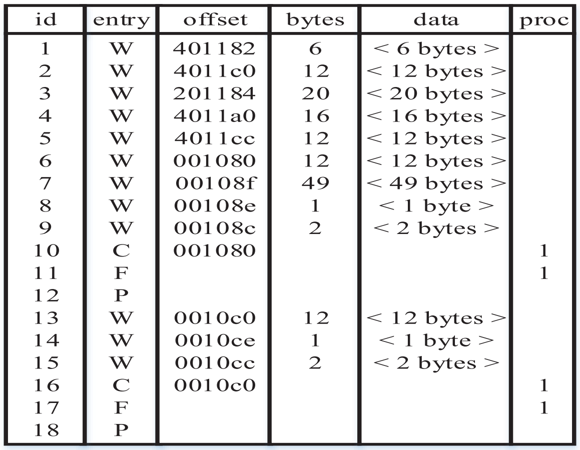
After Yat-replay is done applying the last combination of a segment, it automatically advances to the next segment. At the transition from each segment to the next, Yat-replay removes from the active set any writes that are known to have been made durable by the *pm_wbarrier* separating the two segments. To do this, Yat-replay modifies the PM state by applying these writes without using CoW, so they are made permanent for subsequent tests, thereby advancing App-initial-state. Yat then adds to the active set the writes in the new segment.

Yat-record has the capability of recording additional information in the trace, in the form of *annotations*, to aid in diagnosing failures. In testing PMFS, for instance, we used annotations to indicate what shell command was being executed, and, within PMFS, what transaction type was being performed. When Yat-replay reported a failure, these annotations allowed us to quickly determine what part of the original test was being replayed and what part of the PMFS code was involved. In a trace with hundreds of thousands of entries, it would have been very difficult to diagnose failures without such annotations. Examining the traces (with annotations) collected by Yat-record gave significant insight into the operation of PMFS code, and in general was very helpful in debugging and fixing several bugs.

Figure 3 shows two segments of a sample trace. The columns are as follows:

id an identifier for the trace entry

entry an indication of whether the trace entry is a write, *clflush*, *fence*, or *pm_wbarrier*



id	entry	offset	bytes	data	proc
1	W	401182	6	< 6 bytes >	
2	W	4011c0	12	< 12 bytes >	
3	W	201184	20	< 20 bytes >	
4	W	4011a0	16	< 16 bytes >	
5	W	4011cc	12	< 12 bytes >	
6	W	001080	12	< 12 bytes >	
7	W	00108f	49	< 49 bytes >	
8	W	00108e	1	< 1 byte >	
9	W	00108c	2	< 2 bytes >	
10	C	001080			1
11	F				1
12	P				
13	W	0010c0	12	< 12 bytes >	
14	W	0010ce	1	< 1 byte >	
15	W	0010cc	2	< 2 bytes >	
16	C	0010c0			1
17	F				1
18	P				

Figure 3: Excerpt of App-trace

offset the offset in the PM of the write or *clflush* operation

bytes the number of bytes written

data the data that was written

proc an indication of which processor executed a *clflush* or *fence*

For the purposes of explanation, assume that the *pm_wbarrier* preceding this excerpt made all outstanding writes durable, so the active set is empty at the beginning.

First, the 9 writes at lines 1 – 9 are added to the active set. These writes modify 3 cache lines, 401180, 4011c0, and, 001080. There are 3 writes to the first cache line, 2 writes to the second, and 4 to the third. A total of 59 combinations are generated for this segment: $(3+1) \times (2+1) \times (4+1) - 1$. After each of these 59 combinations is tested, Yat-replay processes the *pm_wbarrier* at line 12. The writes to cache line 001080 are made durable by this *pm_wbarrier*, because of the *clflush* at line 10, which was fenced at line 11. So these writes (lines 6 – 9) are removed from the active set. The other writes (lines 1 – 5) are retained in the active set. Yat-replay then proceeds to the next segment, and adds the 3 writes at lines 13 – 15 to the active set. The active set for this segment contains 8 writes to 3 cache lines, and the number of combinations is 47: $(3+1) \times (2+1) \times (3+1) - 1$.

Optimizations As mentioned before, the number of combinations in the Reorder phase grows exponentially with the number of writes, and can easily become very large. While a configurable threshold for the number of combinations provides a reasonable (probabilistic) compromise, Yat-replay performance is very important for good coverage. We optimized Yat-replay in several ways.

Even though App may be multi-threaded, the replay/test cycle is single-threaded. To increase the amount of testing that can be completed, we run multiple instances of Yat-replay in parallel. Each replay instance has a separate copy of the PM state. All the instances of

Test	writes	clflushes	pm_wbarriers	Segments		Combinations		Time	
				Total	Threshold	Total	Threshold	Total	Threshold
T1	506	372	131	131	12	15K	4K	55m	15m
T2	54K	14K	6K	6K	4K	789M	1M	5.2y	3d
T3	158K	53K	15K	14K	6K	3.7×10^{78}	2M	∞	5d

Table 1: Evaluation of PMFS test coverage with Yat

replay work from a single trace, but each is assigned a different part of the trace to work on. This results in a speed-up factor nearly equal to the number of hardware threads available.

In many places in App, a large number of consecutive writes are performed that have no temporal dependency. For example, when PMFS uses strcpy to fill in a file name, the order of the write of each character of the file name is immaterial. It would be useless (and impossible) for Yat to attempt to test every possible ordering in such a situation. To avoid this, Yat-record detects adjacent writes to a single cache line and coalesces them into a single write entry in App-trace. This also makes the trace more compact. A single write operation in the trace can be up to a full cache line. Note that this optimization can reduce the effectiveness of testing, because it neglects to test some reorderings that might be significant. An analysis of App might help determine whether it contains sequences of writes where this optimization would be unacceptable. Additional heuristics could be applied to avoid doing this in such cases.

App optimizations App is executed during the replay phase only to perform recovery. The normal, optimized code paths in App are not used at all. Under normal conditions, App recovery code would run rarely, if ever, so it would not be a target for optimizations. However, Yat runs App recovery code millions of times, so its performance has a significant effect on the amount of testing that can be completed. We found it worthwhile to track down and eliminate some bottlenecks in App recovery code, such as use of global locks.

5 Evaluation

In this section, we describe our experiences using Yat for validating meta-data consistency in PMFS. PMFS recovery is built into PMFS itself; it attempts to recover on the next mount after an unclean unmount. We implemented a separate consistency checker tool (fsck.pmfs) to check for PMFS meta-data consistency after recovery.

Examples of bugs found in PMFS Yat helped us find several bugs in PMFS journaling and recovering, and was instrumental in testing the correctness of PMFS. The earliest bugs we found using Yat were coding errors in the recovery code. We were able to find these bugs due to Yat's ability to exercise App recovery code in scenarios that are otherwise hard to create.

The second common type of bug that was easily detected by Yat was a failure to log a particular modification to the file system. In PMFS, creating a log entry causes PMFS to track dirty cache lines and later perform a *clflush* of the relevent cache line. If the log call is omitted, the *clflush* is not done, which Yat easily detects. In one case, the fields that were not logged were the access and modification times of a file; an incorrectly updated value in those fields would be difficult to detect in any other way, because the outdated value is within the expected range for the field.

A more complex example of a bug detected by Yat was a case where two inodes were being deleted on two separate threads. PMFS uses a "truncate list" to free the blocks used by a file when recovering from a failure that occurs during deletion. If a failure occurs after the truncate-list transaction is committed, recovery will process the truncate list and perform the steps to truncate the file. Essentially, the truncate list acts as a redo journal.

In the failure scenario, thread 1 starts transaction 1 and deletes inode 1, adding it to the truncate list. Before transaction 1 is committed, thread 2 starts transaction 2 and deletes inode 2, adding it to the truncate list. Transaction 2 completes and is committed, and then a Yat-simulated failure occurs.

During recovery, uncommitted transaction 1 is reverted, including removing inode 1 from the truncate list. However, because the truncate list is a linked list, removing inode 1 from the list also removes inode 2 from the list, so the blocks that had been owned by inode 2 are never freed.

This bug was detected by fsck because the link pointer of inode 2 was not cleared, as it would have been if inode 2 had been on the truncate list to be processed by the recovery code. This link field is only cleared to allow detection of this sort of problem. The cause of the bug was that the truncate list was unlocked before the containing transaction was committed, allowing another thread to modify the list. The solution was to commit the transaction before unlocking the list.

This solution led to another bug, also detected by Yat. If a failure occurs during recovery, some of the steps to truncate the inode and free its blocks may be completed, but the inode is still on the truncate list. During the next attempt at recovery, the inode is in an unexpected state and recovery fails. This bug was detected by Yat in multiple ways, depending on the point of failure during recovery. In one case, the recovery code failed due to a NULL pointer access; in another case, fsck found that the size of the inode was incorrect after recovery. To solve this, the code that processes the truncate list during recovery must not assume anything about the state of the inode and must be idempotent.

Yat performance For the performance evaluation of Yat, we tested PMFS with Yat on a system with 3GHz Intel third generation Core i5 processor and 8GB memory. The processor has 4 cores and 8 threads. The system is running a Linux 3.3 kernel (with PMFS), booted as the Hypersim guest OS.

Table 1 shows sample Yat performance when validating PMFS. The tests were performed using a threshold of 250 for the maximum number of combinations per segment. The Segments-Threshold column is the number of segments where the number of combinations would have exceeded 250. The Combinations-Total column is the total number of combinations for the test if the number of combinations for each segment were unlimited. The Time columns are the approximate times required to run the number of combinations in the Combinations columns. T1 runs 100 simple shell commands to make directory, create a file, and append small amounts of text data to the file. T2 runs 1200 commands in the same mix as above. T3 runs 75 (more complex) commands that copy large files to PMFS and tar the contents.

Replay performance is highly dependent on App and App-chk, as most of the time is spent in the recovery and checking steps. We observed that 5% of the total replay time was spent setting up each test, 65% in App recovery, and 30% in App-chk. Since Yat performance scales almost linearly with available compute power, we can further improve validation time by using more capable (even distributed) systems.

We were able to both detect and diagnose bugs by examining the trace manually. We believe that we can automate some of this. In other words, we can detect some bugs automatically simply by using some fairly simple heuristics to analyze the trace, without running the replay at all. One heuristic that is completely independent of the design of App is that if a cache line has been written to but not clflushed, and thus remains in the active set across a number of segments (10 segments, perhaps), it is likely that the *clflush* was omitted.

6 Related Work

Providing consistency and recovery in the face of failures is challenging, therefore necessitates extensive testing. In its goals, Yat is similar to previous efforts that used a combination of model-based analysis, automation, and knowledge of the application to achieve good test coverage in reasonable time [7].

The possible failure modes for storage software (such as file systems and databases) depends on the characteristics of the underlying storage medium and its interface to the rest of the system. Prior work focused on testing software built for block-based devices with a separate address space [7]. In contrast, PM is a byte-addressable storage medium that resides in the same address space as regular volatile memory. Though the challenge posed by evictions and reordering is similar for PM and block based systems, implementing a permutation-based testing framework for PM requires different techniques and optimizations. To the best of our knowledge, this paper is the first to define and address the challenges to testing software posed by PM.

7 Conclusion and Future Work

Yat, a generic PM software validation framework, has several key characteristics that make it effective for testing and debugging PM software. After capturing a sequences of writes and fence operations by App, Yat tests all permissible orderings of these operations, resulting in extensive coverage of possible error conditions in App. When it detects a failure, Yat reports the exact sequence of operations that led up to the failure, aiding in the diagnosis of the failure. Furthermore, Yat is fast enough that it is practical for use in testing real-world software, as demonstrated in the testing of PMFS.

We are looking at ways to avoid having to modify the PM application source code to cause VM exits for *clflush* and fence instructions and *pm_wbarrier* primitive. Binary translation tools potentially can address both these goals. We are also planning to port Yat from a VMM environment to a OS native environment to debug application level PM code without needing a hypervisor. This should greatly simplify the setup overhead for Yat.

References

[1] Barham, Paul and Dragovic, Boris and Fraser, Keir and Hand, Steven and Harris, Tim and Ho, Alex and Neugebauer, Rolf and Pratt, Ian and Warfield, Andrew. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, ACM.

[2] Dulloor, Subramanya R. and Kumar, Sanjay and Keshavamurthy, Anil and Lantz, Philip and Reddy, Dheeraj and Sankaran, Rajesh and Jackson, Jeff. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, ACM.

[3] Intel. Pin - A Dynamic Binary Instrumentation Tool, 2012.

[4] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, 2013.

[5] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable high performance main memory system using phase-change memory technology. In *In International Symposium on Computer Architecture (ISCA '09).*

[6] Volos, Haris and Tack, Andres Jaan and Swift, Michael M. Mnemosyne: lightweight persistent memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS XVI, ACM.

[7] Yang, Junfeng and Twohey, Paul and Engler, Dawson and Musuvathi, Madanlal. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006).

Medusa: Managing Concurrency and Communication in Embedded Systems

Thomas W. Barr
Rice University

Scott Rixner
Rice University

Abstract

Microcontroller systems are almost always concurrent, event-driven systems. They monitor external events and control actuators. Typically, these systems are written in C with very little support from system software. The concurrency in these applications is implemented with hand-coded interrupt routines. Race conditions and other classic pitfalls of implementing parallel systems in shared-state programming languages have caused catastrophic, and sometimes lethal, failures in the past.

We have designed and implemented Medusa, a programming environment for microcontrollers using the actor model. This paper presents three key contributions. First, the Medusa language, which is derived from Python and Erlang. Second, an implementation that runs on systems several orders of magnitude smaller than any other actor-model system previously described. Finally, a novel bridging mechanism to extend the domain of the actor-model to hardware. Combined, these innovations make it far easier to build complex, reliable and safe embedded systems.

1 Introduction

This paper presents the design, implementation, and evaluation of Medusa, a high-level language for embedded microcontrollers. Medusa integrates hardware and software messaging to provide a robust, easy to use concurrent programming environment ideally suited for small microcontrollers.

Microcontrollers are fundamentally designed to be a part of an event-driven system. They are connected to sensors and actuators and operate in response to external stimuli. Given that microcontrollers are used to control physical systems—such as microwave ovens, cars, and industrial machinery—embedded software must be robust and reliable. However, the event-driven nature of these systems leads to concurrency and synchronization issues that are notoriously difficult to manage, even in large scale systems with system software support [10, 28, 29]. In embedded systems, the programmer is largely unaided in dealing with these complex issues. Designing better programming systems for small scale microcontrollers is becoming critically important as these devices proliferate.

At best, embedded systems utilize a real-time operating system (RTOS) to help manage concurrency. These systems provide primitive, low-level mechanisms for thread scheduling, synchronization, and communication [2, 5, 9, 18]. However, these systems do not directly address the challenges of event-driven systems. Programmers are still responsible for allocating resources to tasks, arbitrating access to peripherals, and synchronizing access to shared data. This is difficult and error-prone.

Medusa is a programming language and run-time system for developing concurrent microcontroller-based systems designed to address these issues. Medusa is based on the actor model of concurrency [15, 1]. The actor model solves the fragmented control flow and shared state problems inherent in traditional approaches to event-driven programming [10]. The Medusa programming model and run-time system utilize and expand upon these ideas for small embedded systems.

This paper presents three practical contributions. First, it presents a new actor-based programming language that is implemented as a small set of extensions to Python, a popular and expressive programming language. Medusa's message passing system is based on Erlang, which has been rigorously evaluated, both formally and in practice. This combination makes Medusa simple enough to be used by a novice and yet expressive enough to build complex concurrent applications.

Second, we present an implementation of the Medusa system as a set of extensions to Owl, our open-source embedded Python implementation [4]. These extensions are very small, less than 3KB of compiled code. This allows Medusa to run on systems that have less than 1%

of the *minimum* memory requirements of Scala or Erlang. An empty thread only consumes 130 bytes, and there is no fixed limit to the number of threads in the system. It is also fast: The time needed to prepare a message, send it from one thread to another and finally process it on the other end is less than the time required for five function calls. The scheduler can both spawn a new thread and perform a context switch in less than the time required for a single function call. Even on a microcontroller with 96 KB of RAM, Medusa can support hundreds of threads. We evaluate the complete system with microbenchmarks and realistic embedded applications, demonstrating that modern language features can be used on systems smaller than any that have been previously demonstrated—or even proposed.

Finally, this paper presents a novel bridging mechanism to extend the domain of the actor model to the hardware. This mechanism is fast; accepting, converting and storing an external event takes less than 4 microseconds. Further, these bridges simplify one of the more difficult challenges of embedded systems software development: dealing with concurrent interrupts. By presenting the exact same abstraction for communication both among software threads and hardware, the programming model is more uniform, making it easier to design and implement robust and reliable embedded systems.

The following section discusses background and related work. Section 3 describes the Medusa language itself. Sections 4 and 5 describe the implementation of the messaging, bridging and toolchain systems of Medusa in detail. Section 6 presents a quantitative analysis along with real-life applications built with Medusa. Finally, we conclude in Section 7.

2 Background and Related Work

Traditionally, microcontrollers are programmed in C. To address the challenges inherent in C programming, a typical microcontroller tools vendor provides a suite of software to help analyze and debug low-level C programs running directly on the microcontroller or on top of a thin RTOS. While these tools help the programmer find problems, they do little to simplify the task of writing and maintaining low-level embedded software.

These tools have evolved to perform static analysis to detect specific, common bugs. One of the most common themes of these systems is the detection of data races [17, 27]. We believe, however, that the best way to prevent these common mistakes is to preclude them in the programming environment. Recent research efforts have started to move in this direction by introducing new, higher-level programming languages and new programming models to small systems. Medusa does exactly this. It extends our open-source managed run-time to directly support the actor model of event driven programming. In the Medusa system, these static analysis tools become unnecessary because *it is impossible for a Medusa programmer to introduce any of these types of bugs.*

2.1 Embedded managed run-time systems

Early commercial embedded run-time systems, such as the BASIC Stamp [20], were quite primitive. As such, they never moved far past educational uses. Similarly, academic projects were largely focused on extremely small 8-bit devices [22].

Since then, microcontrollers have grown in size and capability, so they can support more capable run-time systems. The Java Card environment allows a small subset of Java to run on 16-bit microcontrollers [6]. The recent availability of 32-bit microcontrollers has allowed for the creation of much larger virtual machines, such as Squawk for Java [26] and Owl for Python [4]. The open-source Owl toolchain and virtual machine serves as the base for the Medusa system. In total, the Medusa Virtual Machine consists of 24,200 lines of C.

On high-end embedded systems like phones, managed run-time systems are nearly ubiquitous. The Android system runs on top of Dalvik, a bytecode virtual machine and the iPhone runs on top of the Objective-C 2.0 automatic garbage collection and reflection runtime.

The Erlang system itself has been used on embedded systems such as phone switches and base stations. However, these devices are very different from those targeted by Medusa. The Erlang website[1] states, “People successfully run the Ericsson implementation of Erlang on systems with as little as 16MByte of RAM.” This is over 200 times the minimum requirements of Medusa.

2.2 Actors

The actor model [15, 1] is a mathematical framework that directly represents event-driven systems. Actors represent distinct modules of computation, each responsible for a logical task. They receive a message, send messages to other actors, then decide how to respond to future messages. They do not, however, modify shared state. This structure makes program control-flow much easier to understand. The actor model cleanly separates and isolates system function [10].

The actor model was first described in 1973 by Hewitt, Bishop and Steiger [15] as a framework for artificial intelligence. Early work on actors established a strong mathematical and theoretical basis for the proving and verifying aspects of actor-based systems. These include formal definitions of what an actor system is [7], laws for

[1] http://www.erlang.org/faq/implementations.html

actor systems [14], commitment [13] and formal analysis of divergence and deadlock [1].

Many programming languages have been constructed to directly support the actor model, including Act [24], Erlang [3], Scala [25] and Go.[2] This work has collectively built useful infrastructure for practical actor-based systems, formal verification systems (enabled by the strong theoretical background of functional and actor programming) [16] and efficient systems for generating and passing object references [12]. Many of these languages, most notably Go and Erlang, contain threading implementations that are lighter than operating system threads [3, 11] but are still more resource intensive than Medusa. Go and Scala require more than an order of magnitude more memory than Medusa.

TinyOS is another microcontroller programming environment that proposes a new programming model. In this *split-phase model*, programs are divided into blocks that start long-running operations, but do not wait for them to complete. Instead, they are notified that operations have succeeded through callbacks [23]. This is a potentially difficult-to-use programming model. In fact, more recent research has provided a threading model that runs on top of TinyOS's split-phase model [19].

Scala and Erlang both provide a messaging layer that is similar to Medusa's software-to-software messages; they can transport complex objects and have sophisticated pattern matching abilities. Candygram, a Python library, provides similar facilities. However, it does not include new syntax for messaging and pattern matching nor lightweight threads. On a desktop computer, Candygram threads were measured to require 3.5 KB of memory each, several hundred times what is required in Medusa.[3] The Go language provides a messaging layer that is easy to access from low-level code but does not provide composite object messaging or pattern matching.

3 Medusa language

The Medusa language is a backwards-compatible, extended version of Python that directly supports the actor model. This is analogous to Scala, a similarly extended version of Java. Medusa includes several key features derived from Erlang: light-weight threads, messaging, pattern matching, and atoms.

In Medusa, actors are implemented as light-weight threads. A context switch takes roughly the same amount of time as the execution of a Medusa function call. The interface for creating a new thread in Medusa is extremely simple, using `thread.spawn`. Once the new thread is running, it is also simple to send messages to the spawned thread. Any immutable object can be sent as a message. For example:

```
new_thread = thread.spawn(function)
new_thread.send(1)
new_thread.send((1, "foo", 2.1, True, None))
new_thread.send((1, "foo", 2.1, (1, 2, 3)))
```

```
recv:
    case 1:
        print "received 1"
    case 2:
        print "received 2"
```

Figure 1: Basic Receive Statement.

Once the actor has been started, it receives a message using the `recv` statement, shown in Figure 1. The interpreter matches against each `case` block sequentially. If the message matches the first case, the message will be consumed, the print statement will be executed and the `recv` block will finish. Note that code does not "fall through" into other case blocks. If the message fails to match any block, the message will be deferred for later handling, and the system will wait for a new message.

Often, the programmer will not know the exact message an actor is expected to receive. Medusa allows this through pattern matching, where a portion of the message is specified and other parts are stored as variables:

```
recv:
    case ("fire!", 1, temperature):
       print "engine one on fire!"

    case ("fire!", 2, temperature):
       print "engine two on fire!"
```

A pattern can contain immutable values plus any number of variable names. If a variable is already bound, the message will only match if the value in the message is the same as the value bound to the variable.

When a message is received, each element in the message is compared against the first pattern. For example, assume the above actor is sent the message `("fire!", 1, 1205)`. The first two elements of the message match the pattern. The system then assigns the values from the message to the unbound variable names. In this case `temperature` is assigned the value `1205`. If the pattern does not match all of the bound elements of the message, all unbound variables remain unbound. Literal values, bound variables and unbound variables can appear in any order in a pattern. Additionally, patterns and messages can be arbitrarily complex; patterns are matched with a deep comparison. Finally, the keyword `Any` acts as a wildcard and can be used in a pattern to match any value. This is useful when a part of a message does not need to be saved.

In addition to `recv` blocks, Medusa allows pattern matching with the new match operator (`<-`):

[2] http://golang.org/

[3] http://candygram.sourceforge.net/

```
import thread

def start():
   EchoThd <- thread.spawn(echo)
   EchoThd.send((mytid(), "hello"))
   recv:
      case (EchoThd.tid(), Msg):
         print Msg
   EchoThd.send(‘stop‘)

def echo():
   recv:
      case (FromTid, Msg):
         FromThd <- thread.get_thd(FromTid)
         FromThd.send(mytid(), Msg)
         return echo()
      case ‘stop‘:
         print "Stopping", mytid()
```

Figure 2: Medusa echo program.

```
data = ("baz", 42)
("baz", number) <- data
```

In this example, the pattern `("baz", number)` matches against `data`. The value `42` is stored to the variable `number`. If the pattern matching had failed, however, an exception would have been thrown, stopping the execution of this thread with an error.

In addition to complex patterns, as shown above, the match operator can be used with the trivial pattern: `a <- 32`, pronounced "a gets thirty-two". This enables single assignment, guaranteeing that a variable will never change value, providing support for functional programming.

In the examples so far, strings have been used to distinguish message types. Statically typed languages, such as Scala, use actual object types instead. Dynamically typed languages, such as Erlang, use "atoms". Internally, atoms are handled more efficiently than strings, as they are merely textual representations of a unique identifier. Medusa supports atoms using the backtick delimiter:

```
case (‘fire‘, ‘apu-one‘, temperature):
```

Finally, Figure 2 shows an example that combines all of these elements. A main thread spawns an actor that echoes messages back to the sender. The main thread then sends a message to the echo server, waits for the proper response, then stops the server with the `‘stop‘` atom. After receiving a message, the echo actor recursively calls itself to handle the next message. The Medusa compiler automatically optimizes this tail-call so no stack space is used.

4 Implementation

The Medusa system is implemented as a set of extensions to our previously-published embedded Python system, Owl [4]. Specifically, it is built using a small number of new bytecodes, object types, and a modified Python compiler. These extensions are completely backwards compatible. Existing code runs unmodified. Python and Medusa code can even run at the same time on the same device. They consume 3KB of compiled code, less than 10% of the overall size of our virtual machine. While our implementation was designed around the existing Owl system, our design should be generalizable to implementing lightweight messaging and pattern matching in any bytecode virtual machine.

4.1 Pattern Matching

Pattern matching is integral to the Medusa messaging system. It allows threads to concisely specify the messages that they are ready to receive. Pattern matching is implemented by combining Python's existing comparison support with a new "unbound variable" object. For example, consider this pattern match operation:

```
(12, a, b) <- (12, 3, 4)
```

The Medusa compiler compiles this almost exactly as if it were a standard comparison:

```
Offset  Bytecode            Argument

     0  LOAD_CONST          (12)
     3  LOAD_NAME_UNBOUND   (a)
     6  LOAD_NAME_UNBOUND   (b)
     9  BUILD_TUPLE         3

    12  LOAD_CONST          (12)
    15  LOAD_CONST          (3)
    18  LOAD_CONST          (4)
    21  BUILD_TUPLE         (3)

    24  COMPARE_OP          (<-)
```

For variable loads on the left hand side of the pattern match, the compiler emits `LOAD_NAME_UNBOUND` instead of `LOAD_NAME`. When the program executes `LOAD_NAME_UNBOUND` on the name `a`, it looks up the variable `a` to see if it is bound. If it is, it loads its value and pushes it onto the stack, exactly as `LOAD_NAME` would do. If it is not, instead of raising a name error exception, it creates a new `UnboundLocal` object as a placeholder for the future value for the variable named `a`. Similarly, the variable `b` is looked up, and either its value or a new `UnboundLocal` is pushed on the stack. For the sake of this example, assume that `a` was previously bound to 3 and `b` is unbound.

When execution reaches `COMPARE_OP`, the virtual machine will compare the top two objects on the stack:

```
(12, 3, [UnboundLocal for name "b"])
(12, 3, 4)
```

`COMPARE_OP` for the bind operator starts by creating an empty dictionary to store unbound objects with their new values. Then, it performs a nested comparison on the two tuples just as it would in standard Python.

It starts with the first element, compares it, then moves on. Finally, with the last element, it compares the literal value 4 with the unbound object [UnboundLocal for name "b"]. This comparison always succeeds, since b does not yet have a value. The virtual machine adds an entry to the dictionary associating the name of the unbound local with the value it was compared with. If the value is not going to be used, a special Any object can be used, instead of an unbound variable, to match anything. If every element in the comparison matches, this means that the pattern has matched. At this point, each of the entries in the unbound objects dictionary can be committed into actual variables.

A receive statement consists of a sequence of these patterns. As a message arrives, an attempted match is made against each pattern in the receive statement. If a match fails, the next pattern is tried. If a match succeeds, the variables are bound and the code block associated with that pattern is executed.

4.2 Mailboxes

In Medusa, each thread has a pair of queues to support message passing: one for incoming messages (the mailbox queue) and one for deferred messages (the deferred queue). When a thread sends a message, it appends a reference to the message object in the destination thread's mailbox. Messages cannot contain mutable objects, so they do not need to be copied. When a thread executes the recv statement, the virtual machine first moves any messages from the deferred queue into the front of the mailbox, using the UNDEFER_MSG bytecode. Then, it removes the first message from the mailbox, using the RECV_MSG bytecode, and attempts to match it against each pattern, in order.

If none of the patterns match, the message is appended to the end of the deferred queue, using the DEFER_MSG bytecode. The next message is removed from the mailbox, and the process repeats. Finally, if the mailbox is empty (either because the thread had no pending messages or because all the pending messages were deferred), the thread blocks, waiting for another message. This allows the scheduler to execute other threads.

The key to this process is that when messages from the deferred queue are moved back into the mailbox, they remain in the order of their arrival. If a given message cannot be handled by an actor in its current state, a later message may reconfigure the actor to be able to handle the older message. It can then be handled and removed from the mailbox. This prevents deadlock.

The implementation of the receive process is illustrated in Figures 1 and 3. Figure 1 shows a simple receive block with more than one case. Figure 3 shows the compiled bytecode for this basic receive statement.

```
Offset  Bytecode           Argument

     0  UNDEFER_MSG
     1  RECV_MSG

     2  DUP_TOP
     3  LOAD_CONST         (1)
     6  ROT_TWO
     7  COMPARE_OP         (<-)
    10  POP_JUMP_IF_FALSE  21

    13  LOAD_CONST         ("received 1")
    16  PRINT_ITEM
    17  PRINT_NEWLINE
    18  JUMP_ABSOLUTE      44

    21  DUP_TOP
    22  LOAD_CONST         (2)
    25  ROT_TWO
    26  COMPARE_OP         (<-)
    29  POP_JUMP_IF_FALSE  40

    32  LOAD_CONST         ("received 2")
    35  PRINT_ITEM
    36  PRINT_NEWLINE
    37  JUMP_ABSOLUTE      44

    40  DEFER_MSG
    41  JUMP_ABSOLUTE      1

    44  POP_TOP
```

Figure 3: Bytecodes for code in Figure 1.

The first two bytecodes move all the messages that might have previously been deferred back into the mailbox queue (at the front, and in the order that they arrived) and then receives the first message from the mailbox and places it on the stack. The next block of code (bytecode offsets 2–10) performs the first pattern match against the pattern "1". The message is duplicated, the constant 1 is placed on the stack, and they are swapped (because the message must be on the top of the stack). The COMPARE_OP bytecode then actually does the matching and places True or False on the stack depending on whether the match was successful or not.

If the first match fails, the POP_JUMP_IF_FALSE bytecode skips to the next pattern match (bytecode offsets 21–29), which is nearly identical in this case. If the match succeeded, the pop jump bytecode will simply pop True off the stack and fall through. The associated code (bytecode offsets 13–18) will execute, printing the string "received 1" and jumping to the end of the block (bytecode offset 44) which simply pops the original message off the stack.

If both matches fail, the message will be deferred (bytecode offset 40) and the code will jump back to the receive (bytecode offset 1). If there is another message, it will try to match that message again. If not, the RECV_MSG bytecode will block waiting for the next message to arrive and the thread will yield the processor.

Notice that the bulk of the work here is done by the compiler with only a few specialized bytecodes. This provides significant flexibility to use these mechanisms

```
def monitor1(controller):
    # wait for event 1 to occur
    controller.send(me().tid(), "alert!")
    monitor1(controller)

def monitor2(controller):
    # wait for event 2 to occur
    controller.send(me().tid(), "alert!")
    monitor2(controller)

def init():
    # spawn two monitors to send me alerts
    m1 <- thread.spawn(monitor1, me())
    m2 <- thread.spawn(monitor2, me())

    # process events
    event_loop(m1.tid(), m2.tid())

def event_loop(m1tid, m2tid):
    recv:
        case (m1tid, msg):
            # process alert from m1
        case (m2tid, msg):
            # process alert from m2
        case Any:
            print "Unexpected message!"

    # Always return to the event loop
    event_loop(m1tid, m2tid)
```

Figure 4: Simple Messaging Example.

and ample opportunity for compiler optimization for special cases.

4.3 Example

Figure 4 shows a simple example of how messaging works. In this example, two monitor threads are spawned which wait for arbitrary events. When they receive a message, they send a message back to the main control thread. Here, the messages are simple strings, but they could be arbitrary data. The main control thread runs an event loop waiting for messages from the monitor threads.

When the main control thread receives a message, pattern matching is used to determine what to do. The incoming message is first matched against the pattern `(m1tid, msg)`. In this tuple, the first variable, `m1tid`, is already bound, whereas the second, `msg`, is not. So, this pattern will match any incoming message that is a tuple of two elements with the thread ID of the m1 thread. If this match fails, the next pattern, `(m2tid, msg)`, will be tried. If either of these matches succeed, the `msg` variable will be bound to the data from the second element of the tuple in the message. The last case with the pattern `Any` will match any other message, ensuring the mailbox does not fill with unexpected messages. If information about the unexpected message is desired, the pattern could be a single unbound variable which will match any object (including a tuple). This structure frees the programmer from worrying about the order of message arrival. The loop will process messages as they arrive, and there is a clear and easy way in which to specify how to process each message. Finally, `event_loop` is reinvoked to process the next message. The Medusa compiler optimizes this tail call.

```
def event_loop(m1tid, m2tid):
    recv:
        case (m1tid, msg):
            # process alert from m1

    recv:
        case (m2tid, msg):
            # process alert from m2
        case Any:
            print "Unexpected message!"

    # Always return to the event loop
    event_loop(m1tid, m2tid)
```

Figure 5: Prioritizing Messages.

While this example shows the elegance of the messaging system, mailboxes are designed to provide further control over message receipt to the programmer. Imagine, instead, that it is only useful to process messages from the second monitor after receiving a message from the first monitor. The out-of-order delivery mechanism of the mailbox makes this possible. Consider the revised event loop in Figure 5. In this case, no matter what message arrives first, the main control thread will wait for a message from the first monitor. All other messages, either from the second monitor or unexpected messages, will be deferred.

Once a message from the first monitor is received, it will be processed. The next `recv` block will then be executed. In that case, the first thing that happens is that all received messages that were previously deferred waiting for a message from the first monitor will be “undeferred”. So, if a message from the second monitor had arrived first, it will now be received and processed immediately.

These simple examples demonstrate how the messaging implementation enables flexibility for the programmer. Further, note that the mailbox maintains messages in the order that they arrived. The programmer can reorder these messages only by the structure of `recv` blocks and patterns.

4.4 Deadlock and finite message bounds

The actor model in general is resilient to deadlock. Traditional notions of deadlock, where two threads wait on one another to free locks, are impossible in Medusa since there are no locks. However, it is possible for the system to fail if a large number of messages of one type are sent to a thread that is only receiving a message of a different type. This is true of any actor system with finite message queues [8]. In Medusa, the system will stop with an out-of-memory exception.

The Medusa environment has tools to help diagnose and fix these potential issues. The virtual machine keeps track of the largest number of pending messages that are ever held at once in each thread's mailbox. Additionally, it tracks the average number of pending messages in the mailbox each time the thread receives a message. The programmer can view these values for any threads at any time. If either value is particularly large, there is the potential for mailbox overflow. In that case, the application should probably be restructured.

5 Bridge architecture

Current embedded run-time systems, such as the Owl system that Medusa is based upon, require polling to detect hardware events. Applications must repeatedly check to see if an event has occurred inside peripheral hardware. In contrast, C programs often use interrupts, which requires the use of shared, mutable state.

Medusa introduces a novel third option by *bridging* the domain of the actor model to hardware. Bridges allow interrupt service routines (ISRs) to communicate with software threads. ISRs written using bridges are extremely simple, often just a few dozen lines of C, and all interaction with the virtual machine happens through a safe, secure interface. This interface makes it impossible to introduce a synchronization bug or race condition. With bridges, hardware modules and software threads communicate using the same message passing system.

5.1 Programming with Interrupts

Inside the microcontroller, there are hardware agents that monitor the state of peripherals constantly, even when the core is busy with other work or is sleeping. These agents detect external events that can be selected by the programmer. A user may want to detect level changes on a particular general-purpose I/O (GPIO) port, incoming serial data or a completed analog to digital conversion. When a selected event occurs, the hardware interrupts normal program execution and calls a user function.

Systems written in C use hand-coded interrupt service routines (ISRs) to respond to these events. In effect, this creates a multi-threaded program: one interrupt handler thread and one program thread. The interrupt handler thread must send data back to the program thread so that the program can respond to the event. This must be done by writing to and from shared locations in memory. Since the program thread may be reading or writing from those shared locations, the interrupt and program threads must communicate using thread-safe techniques. These are difficult and error-prone to use, which can lead to catastrophic synchronization bugs. These bugs are unpredictable, only occurring under very specific situations, and can often be virtually impossible to reproduce.

5.2 Interrupt Bridging

Medusa solves these two problems with a technique called *bridging*. It avoids shared state between interrupt handlers and the main program exactly as Medusa solves this problems between multiple software threads: message passing. In effect, bridging makes the hardware agents that monitor for events into actors themselves. These actors run all the time, do not interfere with threads running on the core and send messages when events occur instead of modifying shared state.

Bridges replace the manual process of sharing data with an ISR with a standard, thread-safe interface. With bridging, ISRs are extremely simple and all communication happens through a single function call. Further, the programmer *cannot introduce a synchronization bug or race condition*. Messages from bridges are indistinguishable from other messages and can be received through the `recv` statement, like any other message.

Additionally, the bridge interface used by ISRs is extremely fast. It does not allocate memory off the heap, so it never has to run the garbage collector. The ISR to deliver general-purpose I/O (GPIO) events through a bridge deterministically completes in 187 cycles on our system, less than 4 microseconds. This minimizes the possibility that events will be lost. Overall, bridges eliminate the two biggest challenges of one of the hardest parts of embedded programs: dealing with interrupts.

5.3 Implementation

At its core, a bridge is a producer-consumer ring supporting safe asynchronous communication between two endpoints. In between the endpoints, the virtual machine converts Python or Medusa types to and from C types. With an inbound ring, an ISR calls the bridge code to produce one or more bytes of data. Later, the virtual machine automatically consumes data from this ring and delivers the data to a subscriber. Alternatively, data in an outbound ring is produced by a user thread, then consumed by C code and an interrupt routine.

Each peripheral that will need to either send or receive data from a Medusa thread will have a bridge, denoted by a bridge number, assigned at VM compile time. For each bridge, the user specifies how many bytes each message will contain and how many messages should the ring should hold. For example, consider a bridge to deliver GPIO messages to a Medusa thread. Each event can be described in five bytes, four to represent the 32-bit port number and one to represent the status of all eight bits

```
#define ALL_PINS 0xff
#define MSG_SIZE sizeof(unsigned long) + 1

void GPIOInterruptHandler(unsigned long port)
{
    uint8_t values;
    uint8_t message[MSG_SIZE];

    /* clear all the interrupts for this port */
    GPIOPinIntClear(port, ALL_PINS);

    /* read the value of the port */
    values = GPIOPinRead(port, ALL_PINS);

    /* pack the data into a five byte message */
    memcpy(message, &port, sizeof(unsigned long));
    message[sizeof(unsigned long)] = values;

    /* send it to the subscribers */
    bridge_produce(GPIO_BRIDGE, &message, MSG_SIZE);
}
```

Figure 6: The GPIO interrupt service routine.

on any given port. Therefore, the bridge is set up to send five byte messages:

```
b <- bridge.create(GPIO, # bridge number
                   32,   # number of messages
                   5)    # size of each entry
```

The programmer then sets up the underlying hardware to trigger interrupts on GPIO level transition. Then, the user sets a tag to be included with every message. In this case, the tag is the atom `‘gpio‘` which can be matched by a pattern. Finally, the user specifies the subscriber to the bridge, which will receive any data sent to it. In this case, the subscriber is the current thread:

```
# init the hardware interrupts
interrupt.IntEnable(interrupt.INT_GPIOD)
gpio.GPIOIntTypeSet(button.port,
                    button.pin,
                    gpio.GPIO_BOTH_EDGES)
gpio.GPIOPinIntEnable(button.port,
                      button.pin)

# set the tag and subscriber
b.setTag(‘gpio‘)
b.subscribe(me())
```

The GPIO interrupt handler (Figure 6) constructs these five byte messages and passes them on to the virtual machine for handling. First, it clears the pending interrupt, then reads the GPIO port. It concatenates these values into a single block of size `MSG_SIZE`, then calls `bridge_produce`. This function takes the bridge number, a pointer to the message and the message size.

On the VM side, the system checks that the bridge is initialized and is not full. It then copies the message into the ring. At this point, the interrupt handler returns and the processor resumes executing whatever bytecode it was interpreting when the interrupt occurred.

When that bytecode completes, the virtual machine will consume any new messages out of the bridges by copying them into a Python/Medusa string. The tag specified by the user is combined with the string into a tuple, and that tuple is sent to the subscribing thread. The thread scheduler then marks the subscriber as runnable, and bytecode execution continues. It is critical to realize that this portion of the bridge code is not running inside an ISR, so does not delay or interfere with the execution of other interrupts.

Once the bridge is set up, this process is entirely transparent to the subscriber. That thread simply receives messages, tagged with the specified tag, and acts upon them, just as it would had it received a message from another thread.

5.4 Chronograph

A central timer, or “chronograph”, can also be implemented using bridging. A thread can request that a message be sent to it at some later time. The chronograph then determines which thread will get the next timer wakeup, configures a hardware timer, and waits to receive the message from it. The scheduler automatically suspends the chronograph thread and any threads waiting on a message from it. When the hardware timer goes off, the scheduler wakes up the chronograph, which in turn wakes up and sends a message to the subscribing thread.

The chronograph also provides a `sleep(time)` function that stops the current thread for `time` milliseconds. Internally, this function uses the same hardware timer and bridge as the rest of the chronograph. Using this facility, a thread that is sleeping is automatically suspended until it is time to wake up.

6 Evaluation

To evaluate the Medusa system, we constructed and measured the performance of several microbenchmarks and several embedded applications. We compare applications that use the Medusa systems of messaging and bridging with procedural, polling-based applications that run on the standard Owl system. These experiments were run on the Texas Instruments Stellaris LM3S9B92, an ARM Cortex-M3 microcontroller operating at 50 MHz, with 96 KB of SRAM and has 256 KB of flash.

6.1 Threads

In Medusa, thread state is all stored within the heap. A thread consists of a thread id, message queues, the thread’s activation records, and the thread state (active, blocked, etc.). There is no stack in the system, so all activation records are allocated on the heap and contain a “back” pointer to the previous record. This architecture allows for extremely lightweight thread creation, deletion, and scheduling. On average, it takes about 59*us* to

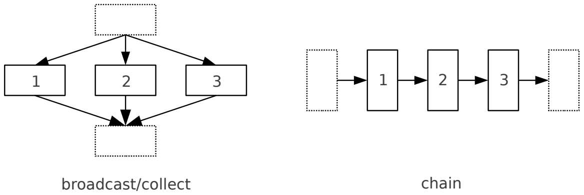


Figure 7: The two message benchmarks: broadcast/collect and chain. The number of workers (3 in the figure) is configurable.

create a thread on a 50 MHz Cortex-M3 processor. It also makes context switches quickly: about 129*us* on average to context switch while running 100 threads.

6.2 Messaging performance

To measure the speed of message passing, two simple benchmarks were constructed, shown in Figure 7. Both spawn a configurable number of worker nodes that receive a message, then forward it on to a specified node. In the broadcast/collect benchmark, a broadcaster node transmits one message to each worker node. They then forward their message to a single collector node. In the chain benchmark, the head node transmits only to the first worker node. That node forwards it on to the second worker node and so on. Finally, the last worker node forwards the message to the collector node.

For both benchmarks, the time between the first and last message sent is timed. These results are shown in Figure 8, normalized to the number of messages sent in each benchmark. At light load, the time required to prepare and send a message, switch contexts to the recipient and finally receive the message is around 600 microseconds for both benchmarks. This is less than the time required for five function calls. As memory pressure increases with more threads, this time increases to around 1000 microseconds. The additional threads create more garbage to be collected, slowing overall progress through the program. The broadcast/collect benchmark is more complex in implementation, so it runs into memory pressure somewhat earlier.

6.3 I/O Latency

The simplest I/O benchmark for Medusa monitors an input pin for a signal and raises an output pin in response. The baseline for this test uses polling. It spins in a loop, reading the input line, writing its state to the output line and yielding back to the scheduler. The other version of this program uses bridges and interrupts. When the pin changes state, an interrupt triggers the bridge interrupt handler described in Section 5.3. It sends a message to a thread, which then raises the output pin. This experiment was run using both standard and priority bridges. These use a modified version of the scheduler that executes a blocked thread immediately when it receives a message from a bridge. To compete for time, the microcontroller runs a heapsort benchmark repeatedly in a background thread. I/O latency was measured using another Stellaris microcontroller and confirmed with a factory-calibrated Tektronix TDS 220 digital oscilloscope. Both benchmarks were tested 500 times each. The distribution of these response times are plotted in Figure 9.

In the polling implementation, the thread reading the input must wait until it is scheduled before it can detect a change. The probability that the GPIO will happen across the 10 ms that the background thread is running is uniform. However, approximately 25% of the time, the background thread triggers the garbage collector, which can last upwards of 85 ms. Once the garbage collector has finished, the polling thread will run and can respond to the input.

The behavior is similar with the standard bridge benchmark. The bridge delivers a message to the I/O thread, which will receive the I/O message once it is scheduled. Again, the garbage collector may interrupt the background thread, slowing response. Note, however, that the bridge implementation will not lose events unless its buffer fills. The bridge interrupt handler will still run during garbage collection, queueing messages. Compare this to polling, which can miss short events during garbage collection. This effect becomes more dramatic as more threads are active in the system competing for processor resources. The best performance comes with the priority bridge. Here, the I/O thread does not have to wait for the background thread to complete its timeslice, so latency is less than 2 ms 80% of the time.

6.4 Streaming data

Many peripherals such as GPS receivers and ultrasonic rangefinders send periodic updates as bursts of data over a serial port. In a GPS unit, update messages are sent once per second in messages ranging from twenty to eighty bytes in size. This presents a problem for applications that use polling to monitor events. While the hardware serial port (UART) on the microcontroller does have a buffer, it too small to hold a complete message. If a program is not polling when a message is sent, the hardware buffer will overflow and the user will not receive the complete message. Additionally, the user program must be able to pull bytes out of the UART buffer at least as fast as they are being received.

Figure 10 shows this behavior. A transmitter sends 50 byte bursts to a receiver. The receiver records the bytes either using a polling loop or with bridges. The first experiment (solid line) shows that even when the controller

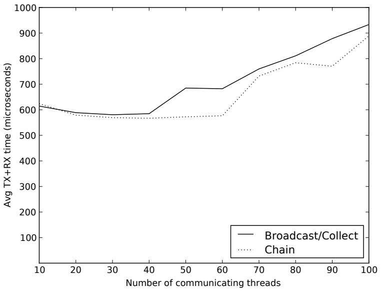


Figure 8: Average time to send and receive a message for different benchmarks.

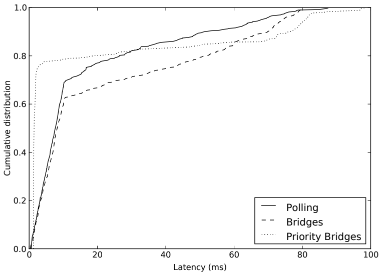


Figure 9: Distribution of I/O latency for polling and bridges.

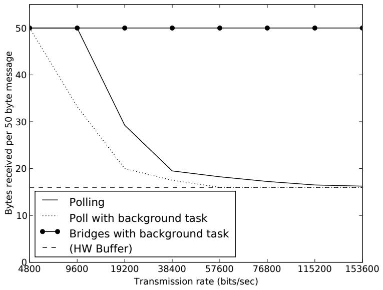


Figure 10: Data loss rates for different transmission rates.

is doing nothing but polling the UART, the software cannot keep up with the transmission rate after 9600 bits/sec. When the polling loop has to compete for time with a background task, it loses data after 4800 bits/sec.

Compare this to an application that receives bytes using bridges and interrupt handlers. In this case, whenever traffic is received, the interrupt handler places the data in a bridge for the receiving thread. Then, that thread is activated by the scheduler and can pull the data from its message queue. In this case, the background task is never interrupted when data is not being sent, and the application can receive the complete 50 byte message at any tested transmission rate.

Similar results are seen with real-life applications. We constructed a simple example that receives and parses data from a GPS receiver. An application using polling could receive and interpret messages up to a transmission rate of 9600 baud. When the GPS transmitted any faster, all messages received are incomplete, and therefore unusable. When the polling thread runs in competition with another thread, no complete messages were received at any data rate. However, with bridges, the GPS can run at its full speed (57600 baud) even with a background thread. Since the GPS thread only runs when it is actively processing a message, the background thread ran at 93% of its native speed.

6.5 Event-driven Systems

The previous sections have shown the efficacy of the Medusa mechanisms for communication and concurrency. The ultimate motivation for these mechanisms is to build better structured event-driven systems. To demonstrate that these mechanisms are both practical and usable in such systems, we built several embedded applications, including an autonomous car and a traffic-light controller.

The autonomous car demonstrates that Medusa can coordinate concurrent tasks and peripherals into a cohesive embedded system. The electronics from an off-the-shelf RC car (an Exceed RC Electric SunFire Off-Road Buggy) were replaced with a 9B92 microcontroller and associated peripherals. The car is controlled entirely by the microcontroller. An ultrasonic range finder, GPS receiver, and three-axis gyroscope connected to the microcontroller transmit feedback from the car's surroundings, while connections to the car's motor and steering servo provide control of the car's movements.

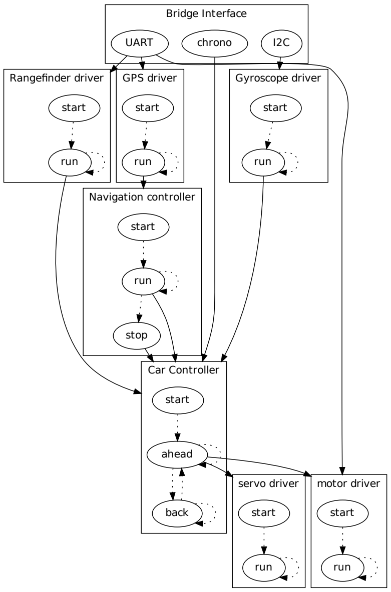


Figure 11: Diagram of Event-driven Autonomous Car.

Figure 11 shows the design of the car application. Each box in the figure represents an actor (Medusa thread). Each oval represents an actor state (Medusa function). Dotted arrows represent state transitions within an actor. Solid arrows represent messages being sent from one actor to another.

The GPS connects to the microcontroller over a UART. Every second, the GPS sends a packet of data that contains control information and its current position.

After each byte of the packet is sent, the microcontroller triggers an interrupt. This signal is caught by the specific UART's bridge interrupt controller, converting the byte from the GPS into a software message. A GPS driver thread subscribes to these messages being sent from the GPS via the UART bridge. When it has received an entire message, it converts the message from a string of bytes into a Medusa message with numeric longitude and latitude. These messages are sent to a navigation controller. This thread maintains a list of waypoints and sends turn commands to the master thread described below.

The motor controller and rangefinder also connect to the microcontroller via UARTs, using a similar combination of bridges and driver threads. The gyroscope connects over the I2C bus, also through a bridge to its own driver module. Finally, the application uses the chronograph to wait for an exact period of time without spinning to read the clock.

The car is controlled by a master thread that collects messages from the GPS, rangefinder and gyro and sends messages to the servo and motor controller threads. This master module runs a two-term, PI feedback controller that makes sure the car drives straight, even over varying terrain (or the author's foot). Meanwhile, it receives turn commands from the navigation thread. Finally, it monitors the rangefinder to avoid hitting obstacles.

Note that the nine threads in this system have no shared state whatsoever. All communication is done through messages. Furthermore, all communication from the hardware takes place through bridges. This means that none of the peripherals are polled. Each driver thread waits to receive a message from its interrupt bridge and is descheduled when there is no data to process. When new data is available, it is rescheduled and execution continues.

During execution, this application sends an average of 315 messages per second, 243 of which come from interrupt bridges and 72 that come from other software threads. The system is idle 52.4% of the time, i.e. all threads are waiting on data from external sources. This allows all messages to be dealt with promptly. On average, there are less than 1.1 messages queued whenever a thread receives a message. The maximum number of messages queued in a thread's mailbox at one time was 40, in the thread that receives bytes from the GPS receiver. These 40 bytes correspond to a single GPS sentence which was likely received while the VM was busy with an uninterruptable task like garbage collection. As soon as that task finished, the GPS driver thread resumed and read the entire message.

This example was originally written as a single-threaded, event-loop based program in Python. While conceptually simple, the concurrent nature of the peripherals proved to be very difficult. The event-loop has to run very slowly and be tuned very carefully to ensure that the control loop has consistently updated data and that input events are not missed. Adding a feature becomes harder as the program grows more complex. Suppose the programmer wants to trigger a periodic event. On program start, a global variable is set storing the next time the event needs to happen. Each time through the control loop, the clock is polled and compared against that global variable. If it is time, the programmer executes the event and resets the global variable. Each feature like this slows down the event loop, degrading performance of everything else in the system. Moreover, if anything else in the event loop takes a long time, the periodic event will be triggered late. Writing the program in C using interrupts and locks would be even more difficult and extremely error-prone due to the large number of events coming from both hardware and software components that need to be synchronized.

The Medusa program is comparatively simple. A periodic task can be implemented by calling the chronograph's sleep function (see Section 5.4), performing the task, and repeating:

```
def periodic():
    chrono.sleep(5000)
    do_task()
    return periodic()

thread.spawn(periodic)
```

Since the chronograph uses hardware timers that do not need to be polled, this task *does not impact others* while it it waiting to run. The many components in a large program just wait for data to be available, process it, and send it on to other components. The bridge, messaging and scheduling systems synchronize everything automatically.

Other demonstration applications have also been built using Medusa such as a distributed traffic-light controller that runs on a microcontroller and a multi-threaded web server that runs on a port of Medusa to a standard x86 Linux system. These examples show that the Medusa system is flexible and easily programmable.

7 Conclusions

Building reliable embedded systems has long been a challenging endeavor. In the 1980s, the Therac-25, a radiation therapy device, suffered several failures due to a race condition between the main program and a hand-coded interrupt service routine. As a result, at least six patients were overdosed and three died. [21] Unfortunately, not much has changed since then. Programmers and industry regulators have been more careful with safety-critical systems, but accidents continue. A race condition in a GE Energy power control system[4] caused widespread blackouts in 2005. More recently, the suspect systems in the Toyota unintended acceleration case used shared-state and hand-coded interrupt handlers. [5]

Medusa is designed specifically to advance the state-of-the-art in this area. Pure functional programming with an efficient messaging system leads to much simpler concurrent programming. That is why such systems are used everywhere from telephone switches to the Facebook messaging platform. Medusa demonstrates that it is possible to take the best ideas from these systems and implement them on resource-constrained systems. Further, Medusa is the first embedded language run-time system capable of integrating hardware interrupts into the software messaging system in a seamless manner.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] T. N. B. Anh and S.-L. Tan. Real-time operating systems for small microcontrollers. *IEEE Micro*, 29(5), 2009.

[3] J. Armstrong. Erlang - a survey of the language and its industrial applications. In *In P. symposium on industrial applications of Prolog (INAP96)*, 1996.

[4] T. W. Barr, R. Smith, and S. Rixner. Design and implementation of an embedded python run-time system. In *P. 2012 USENIX ATC*, 2012.

[5] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob. The performance and energy consumption of embedded real-time operating systems. *IEEE Trans. Computers*, 52(11), 2003.

[6] Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, Boston, MA, USA, 2000.

[7] W. D. Clinger. Foundations of actor semantics. Technical report, Cambridge, MA, USA, 1981.

[8] E. D'Osualdo, J. Kochems, and C. H. L. Ong. Automatic verification of erlang-style concurrency, 2013.

[9] J. Gannssle. The challenges of real-time programming. *Embedded System Programming Magazine*, 2007.

[10] P. Haller and M. Odersky. Event-based programming without inversion of control. In *In Proc. Joint Modular Languages Conference (2006), Springer LNCS*. Springer, 2006.

[11] P. Haller and M. Odersky. Actors that unify threads and events. In *P. 9th international conference on Coordination models and languages*, COORDINATION'07, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *P. 24th European conference on Object-oriented programming*, ECOOP'10, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] C. Hewitt. Coordination, organizations, institutions, and norms in agent systems ii. chapter What Is Commitment? Physical, Organizational, and Social (Revised). Springer-Verlag, Berlin, Heidelberg, 2007.

[14] C. Hewitt and H. G. Baker. Laws for communicating parallel processes. In *IFIP Congress'77*, 1977.

[15] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *P. 3rd international joint conference on Artificial intelligence*, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[16] F. Huch. Verification of erlang programs using abstract interpretation and model checking. In *P. fourth ACM SIGPLAN international conference on Functional programming*, ICFP '99, New York, NY, USA, 1999. ACM.

[17] D. Jacobs and A. Langen. Static analysis of logic programs for independent and parallelism. *J. Log. Program.*, 13(2-3), July 1992.

[18] D. Kalinsky. Basic concepts of real-time operating systems. *LinuxDevices Magazine*, 2003.

[19] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-Elefteri, P. Levis, A. Terzis, and R. Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *SenSys*, volume 9, pages 127–140, 2009.

[20] C. Kuhnel and K. Zahnert. *BASIC Stamp: An Introduction to Microcontrollers*. Newnes, Woburn, MA, USA, 2000.

[21] N. Leveson and C. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7), july 1993.

[22] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *P. 10th ASPLOS*, 2002.

[23] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[24] H. Lieberman. Thinking about lots of things at once without getting confused. Technical Report 626, Massachusetts Institute of Technology Artificial Intelligence Laboratory, May 1981.

[25] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausane, 2004.

[26] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk java virtual machine. In *P. 2nd international conference on Virtual execution environments*, VEE '06, New York, NY, USA, 2006. ACM.

[27] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. Softw. Eng.*, 6(3), May 1980.

[28] W. Wulf and M. Shaw. Global variable considered harmful. *SIGPLAN Not.*, 8(2), Feb. 1973.

[29] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *P. 9th USENIX conference on Operating systems design and implementation*, OSDI'10, Berkeley, CA, USA, 2010. USENIX Association.

[4] http://www.securityfocus.com/news/8412

[5] http://www.nhtsa.gov/UA/

Reliable Writeback for Client-side Flash Caches

Dai Qin
University of Toronto

Angela Demke Brown
University of Toronto

Ashvin Goel
University of Toronto

Abstract

Modern data centers are increasingly using shared storage solutions for ease of management. Data is cached on the client side on inexpensive and high-capacity flash devices, helping improve performance and reduce contention on the storage side. Currently, write-through caching is used because it ensures consistency and durability under client failures, but it offers poor performance for write-heavy workloads.

In this work, we propose two write-back based caching policies, called write-back flush and write-back persist, that provide strong reliability guarantees, under two different client failure models. These policies rely on storage applications, such as file systems and databases, issuing write barriers to persist their data reliably on storage media. Our evaluation shows that these policies perform close to write-back caching, significantly outperforming write-through caching, for both read-heavy and write-heavy workloads.

1 Introduction

Enterprise computing and modern data centers are increasingly deploying shared storage solutions, either as network attached storage (NAS) or storage area networks (SAN), because they offer centralized management and better scalability over directly attached storage. Shared storage allows unified data protection and backup policies, dynamic allocation, and deduplication for better storage efficiency [2, 8, 16, 20].

Shared storage can however suffer from resource contention issues, providing low throughput when serving many clients [12, 20]. Fortunately, servers with flash-based solid state devices (SSD) have become commonly available. These devices offer much higher throughput and lower latency than traditional disks, although at a higher price point than disks [13]. Thus many hybrid storage solutions have been proposed that use the flash devices as a high capacity caching layer to help reduce contention on shared storage [6, 7, 19].

While server-side flash caches improve storage performance, clients accessing shared storage may still observe high I/O latencies due to network accesses (at the link level and the protocol level, such as iSCSI) compared to clients accessing direct-attached storage. Attaching flash devices as a caching layer on the client side provides the performance benefits of direct-attached storage while retaining the benefits of shared storage [2, 4]. Current systems use write-through caching because it simplifies the consistency model. All writes are sent to shared storage and cached on flash devices before being acknowledged to the client. Thus, a failure at the client or the flash device does not affect data consistency on the storage side.

While write-through caching works well for read-heavy workloads, write-heavy workloads observe network latencies and contention on the storage side. In these commonly deployed workloads [8, 18], the write traffic contends with read traffic, and thus small changes in the cache hit rate may have significant impact on read performance [7]. Alternatively, with write-back caching, writes are cached on the flash device and then acknowledged to the client. Dirty cached data is then flushed to storage when it needs to be replaced. However, write-back caching can flush data blocks in any order, causing data inconsistency on the storage side if a client crashes or if the flash device fails for any reason.

Koller et al. [7] propose a write-back based policy, called ordered write-back, for providing storage consistency. Ordered write-back flushes data blocks to the shared storage system in the same order in which the blocks were written to the flash cache. This write ordering guarantees that storage will be consistent until some time in the past, but it does not ensure durability because a write that is acknowledged to the client may not make it to storage on a client failure.

Furthermore, the consistency guarantee provided by the ordered write-back policy depends on failure-free operation of the shared storage system. The problem is that the write ordering semantics are not guaranteed by the block layer of the operating system [1], or by physical devices on the storage system [9], because the physical devices themselves have disk caches and use write-back caching. On a power failure, dirty data in the disk cache may be lost and the storage media can become inconsistent. To overcome this problem, physical disks provide a cache flush command to flush dirty buffers from the disk cache. This command enables implementing barrier semantics for writes [22], because it waits until all dirty data is stored durably on media. The ordered write-back policy would need to issue an expensive barrier operation on each ordered write to ensure consistency. In essence, simple write ordering provides neither durability, nor consistency without the correct use of barriers.

We propose two write-back caching policies, called *write-back flush* and *write-back persist*, that take advantage of write barriers to provide both durability and consistency guarantees in the presence of client failures and power failure at the storage system. These policies rely on storage applications (e.g., file systems, applications on file systems, databases running on raw storage, etc.) issuing write barriers to persist their data, because these barriers are the only reliable method for storing data durably on storage media. For example, journaling file systems issue a barrier before committing a transaction, and applications invoke the `fsync(fd)` system call to flush all file data associated with the `fd` file descriptor. Write-back caching policies only need to enforce reliability guarantees at these barriers, since applications receive no stronger guarantees from the storage system. Our caching polices target files that are read and written on a single client, such as files accessed by a virtual machine (VM) running on the client. Thus, we do not consider coherence between client-side caches.

Our two caching policies are designed to handle two different client failure models that we call *destructive* and *recoverable* failure. Destructive failure assumes that the cached data on the flash device is unrecoverable, because either it is destroyed or there is insufficient time for recovering data from the device. This type of failure can occur, for example, due to flash failure or a fire at the client. Recoverable failure is a weaker model that assumes that the client is unavailable either temporarily or permanently, but the cached data on the flash device is still accessible and can be used for recovery. This type of failure can occur due to a power outage at the client.

The write-back flush caching policy is designed to handle destructive failure. When a storage application issues a barrier request, this policy flushes all dirty blocks cached on the client-side flash device to the shared storage system and then acknowledges the barrier. This policy provides durability and consistency because applications are already expected to handle any storage inconsistency caused by out-of-order writes that may have reached storage between barriers (e.g., undo or ignore the effect of these writes). The main overhead of the write-back flush policy, as compared to write-back caching, is that barrier requests may by delayed for a long time, thus affecting sync-heavy workloads.

The write-back persist caching policy is designed to handle recoverable failure. When an application issues a barrier request, this policy persists the in-memory cache metadata to the client-side flash device atomically. The cache metadata consists of mappings from storage block locations to flash block locations; it is needed to locate blocks on the flash device. Durability and consistency are provided by this policy, assuming that the flash device is still available on a failure, because the cache metadata on the device enables accessing a consistent snapshot of data at the last barrier. This policy has minimal overhead on a barrier because persisting the cache metadata to the flash device is a fast operation.

Our evaluation of the two caching policies shows the following results: 1) both policies perform as well as write-back for read-heavy workloads, 2) the write-back flush policy performs significantly better than write-through for write-heavy workloads, even though it provides the same reliability guarantees, 3) the write-back persist policy performs as well as write-back for write-heavy workloads, even though it provides much stronger reliability guarantees, 4) the write-back persist policy has significant benefits as compared to write-through or write-back flush for sync-heavy workloads.

We make the following contributions: 1) we take advantage of the write barrier interface to greatly simplify the design of client-side flash caching policies, providing both durability and consistency guarantees in the presence of destructive and recoverable failure, 2) we discuss various design optimizations that help improve the performance of these policies, and 3) we implement these policies and show that they provide good performance.

The rest of this paper is organized as follows. We discuss prior work on write-back flash caching in Section 2, providing motivation for our work. Section 3 describes our caching policies and Section 4 describes the design of our caching system and the optimizations that improve the performance of our policies. Section 5 shows the results of our evaluation. Section 6 describes related work and Section 7 presents our conclusions and future work.

2 Motivation

Current client-side flash caches use write-through caching [2, 4] because the client and the client-attached flash are considered more failure prone. This caching method also simplifies using existing virtual machine technology since guest state is not tied to a particular client. Write-through caching trivially provides durability and consistency on destructive client failures because storage is always up-to-date and consistent. However, write-through caching by itself doesn't provide any guarantees on storage failures, unless application-issued barriers are honored on the storage side. The main drawback of write-through caching is that it has high overhead for write-heavy workloads.

Next, we discuss two write-back policies that aim to reduce this overhead.

Ordered Write-back Caching Ordered write-back caching flushes data blocks to storage in the same order in which the blocks were written to the flash cache, thus ensuring point-in-time consistency on a destructive failure [7]. This approach does not provide durability because a write that is acknowledged to an application may never reach storage on a destructive failure.

However, durability is a critical concern in many environments. Consider a simple ordering system in which customers place an order and the system stores the order in a file. After confirming the order with the customer, the system persists the contents of the file by invoking the `fdatasync()` system call, and then notifies the customer that the order has been received.

```
int fd = open(...);
...
write(fd, your_order);
fdatasync(fd);
printf("We have received your order.");
```

The `fdatasync()` system call requires writing file contents to storage media durably and thus the file system issues a write barrier. However, with ordered write-back caching, `fdatasync()` is ignored, and data cached on the flash device may not be available on storage after destructive failure. As a result, recent writes may be lost even though the customer is informed otherwise.

Another serious issue with ignoring barriers is that point-in-time consistency can only be guaranteed under failure-free storage operation, since the storage can cache writes and issue them out of order. To avoid this problem, a barrier needs to be issued on every write on the storage side. Thus simple write ordering for ensuring consistency is both expensive, and unnecessary, as described later.

Journaled Write-Back Caching Koller et al. present a second caching policy called journaled write-back that improves performance over ordered write-back by coalescing writes in the cache [7]. Journaled write-back provides point-in-time consistency guarantees at a system-defined epoch granularity. Within an epoch, writes to the same location can be coalesced on the client. All writes within an epoch are written to a write-ahead log (journal) on the storage side, so that data can be committed atomically to storage at epoch granularity.

Although the paper does not mention it, this approach also requires issuing barriers at commit. The system-defined epoch granularity presents a trade-off, with frequent commits affecting performance, and infrequent commits risking more data loss. Furthermore, the system assumes that sufficient NVRAM is available on the storage side to avoid the overheads of journaling.

Unlike either ordered or journaled write-back caching, our write-back policies ensure durability by taking advantage of application-specified barriers. Also, we do not require any journaling on the storage side because applications have no reliability expectations between barriers.

3 Caching Using Barriers

Storage applications that require consistency and durability already implement their own atomicity scheme (e.g., atomic rename, write-ahead logging, copy-on-write, etc.) or durability scheme (e.g., using `fsync`) via write barriers. Our key insight is that write-back caching policies can efficiently provide both durability and consistency by leveraging these application-specified barriers. Since applications have no storage reliability expectations between barriers, the caching policies also only need to enforce these properties at barriers.

We assume that the client flash cache operates at the block layer (i.e., it is below the client buffer cache and independent of it) and caches data for the underlying shared storage system. We now describe the semantics of write barriers, and then describe our caching policies.

3.1 Write Barriers

The block-level IO interface is typically assumed to consist of read and write operations. However, a write operation to storage does not guarantee durability. In addition, multiple write operations are not guaranteed to reach media in order. All levels of the storage stack, including the block layer of the client or the storage-side operating system, the RAID controller, and the disk controller, can reorder write requests. Modern storage systems complicate the block interface further by allowing IO operations to be issued asynchronously and queued [21].

Durability and write ordering are guaranteed only after a cache flush command is issued by a storage application, making this command a critical component of the block IO interface. The cache flush command is supported by most commonly used storage protocols, such as ATA and SCSI, and is widely used by storage-sensitive applications, such as file systems, databases, source code control systems, mail servers, etc.

The cache flush command ensures that any write request that has been *acknowledged* by the device before a cache flush command is issued is durable by the time the flush command is acknowledged. The status of any write request acknowledged after the flush command is issued is unspecified, i.e., it may or may not be durable after the flush. However, the durability of this acknowledged write will be guaranteed by the next flush command.

The flush command enables the implementation of write barriers to ensure ordering and durability [22]. In particular, applications can issue writes concurrently

Policy	**Recoverable Client Cache Failure**		**Destructive Client Cache Failure**		**Storage Failure**	**Latency**	
	Consistency	Durability	Consistency	Durability	Consistency & Durability	Write	Barrier
Write-through	Yes	Yes	Yes	Yes	Yes[1]	High	Low
Write-back flush	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Low**	**High**
Ordered write-back[2]	Yes	No	Yes	No	No	Low	Low[3]
Write-back persist	**Yes**	**Yes**	**No**	**No**	**Yes**	**Low**	**Low**
Write-back	No	No	No	No	No	Low	Low

Yes[1]: The write-through policy handles storage failure when barriers are supported.
Ordered write-back[2]: Ordered and journaled write-back (proposed in previous work [7]) have the same properties.
Low[3]: Barriers are ignored and hence they don't introduce any additional latency.

Table 1: Comparison of Different Caching Policies

when no ordering is needed. To ensure these writes are all durable, the application waits for these writes to be acknowledged and then issues the cache flush command. When this command is acknowledged, further writes can be issued with the guarantee that they will be ordered after the previous writes.

3.2 Caching Policies

Our caching design is motivated by a simple principle: the caching device should provide *exactly* the same semantics as the physical device, as described in Section 3.1. This approach has two advantages. First, applications running above the caching device get the same reliability guarantees as they expect from the physical device, without requiring any modifications. Second, the caching policies can be simpler and more efficient, because they need to make the minimal guarantees provided by the physical device.

In this section, we present our write-back flush and write-back persist policies. Both policies essentially implement the semantics of the write barrier. The flush policy handles destructive failures in which the flash device may not be available after a client failure, while the persist policy handles recoverable failures in which the flash device is available after a client failure. The choice of these policies in a given environment depends on the type of failure that the storage administrator is anticipating.

3.2.1 Write-Back Flush

The write-back flush policy implements barrier semantics on a cache flush request by flushing dirty data on the flash device to storage. The flush process sends these blocks to storage, issues a write barrier on the storage side, and then acknowledges the command to the client.

Similar to write-through, the write-back flush policy does not require any recovery after failure. Any dirty data cached either on flash or storage since the last barrier may be lost, but it is not expected to be durable anyway. As a result, this policy is resilient to destructive failure.

The main advantage of this approach over write-through caching is that writes have lower latency because dirty blocks can be flushed to storage asynchronously. Moreover, it provides stronger guarantees than vanilla write-through caching because it handles storage failures as well (see Section 2). Compared to write-back caching, barrier requests will take longer with this policy, which primarily affects sync-heavy workloads.

3.2.2 Write-back Persist

The write-back persist policy implements barrier semantics on a cache flush request by atomically flushing dirty cache metadata to the flash device. The dirty file-system blocks are already cached on the flash device, but we also need to atomically persist the cache metadata in client memory to flash, to ensure that blocks can be found on the flash device after a client or storage failure. This metadata contains mappings from block locations on storage to block locations on the flash device, helping to find blocks that are cached on the device.

The write-back persist policy assumes that the flash device is available after failure. During recovery, the cache metadata is read from flash into client memory. The atomic flush operation at each barrier ensures that the metadata provides a consistent state of data in the cache, at the time the last barrier was issued.

The main advantage of write-back persist is that its performance is close to that of ordinary write-back caching. Some latency is added to barrier requests, but persisting the cache metadata to the flash device has low overhead, given the fairly small amount of metadata needed for typical cache sizes. The drawback is that destructive failure cannot be tolerated because large amounts of dirty data may be cached on the flash device, similar to write-back caching. Furthermore, if the client fails permanently, then recovery time may be significant because it will involve moving data from flash using either an out-of-band channel (e.g., live CD), or by physically moving the device to another client.

Operation	Description
`find_mapping`	find a mapping entry in either the clean or the dirty map
`insert_mapping, remove_mapping`	insert or remove mapping in the clean or dirty map
`persist_map`	atomically persist the dirty map to flash
`alloc_block, free_block`	allocate or free a block on flash
`evict_clean_block`	evict a clean block by freeing the block and removing its mapping

Table 2: Mapping and Allocation Operations

Table 1 provides a comparison of the different caching policies. The caching policies are shown in increasing order of performance, with write-through being the slowest and write-back being the fastest caching policy. The write-back flush policy provides the same guarantees as write-through, with low write latency, but with increased barrier latency. The write-back persist policy provides performance close to write-back, but unlike the write-back flush policy, it doesn't handle destructive failure.

4 Design of the Caching System

We now describe the design of our caching system that supports the write-back flush and persist policies. We first present the basic operation of our system, followed by our design for storing the cache metadata and the allocation information. Last, we describe our flushing policy.

4.1 Basic Operation

Our block-level caching system maintains mapping information for each block that is cached on the flash device. This map takes a storage block number as key, and helps find the corresponding block in the cache. We also maintain block allocation and eviction information for all blocks on the flash device. In addition, we use a flush thread to write dirty data blocks on the flash device back to storage. The mapping and allocation operations are shown in Table 2. As we discuss in Section 4.2.2, we separate the mappings for clean and dirty blocks into two maps, called the clean and dirty maps.

Table 3 shows the pseudocode for processing IO requests in our system, using the mapping and allocation operations. On a read request, we use the IO request block number (`bnr`) to find the cached block. On a cache miss, we read the block from storage, allocate a block on flash, write the block contents there, and then insert a mapping entry for the newly cached block in the clean map. On a write request, instead of overwriting a cached block, we allocate a block on flash, write the block contents there, and insert a mapping entry in the dirty map. This no-overwrite approach allows writes to occur concurrently with flushing – a write is not blocked while a previous block version is being flushed. Mappings are updated only after writes are acknowledged to maintain barrier semantics (see Section 3.1).

We avoid delaying read and write requests when the cache is full (which it will be, after it is warm) by only evicting clean blocks from the cache, using the standard LRU replacement policy. The clean blocks are maintained in a separate clean LRU list to speed up eviction. We ensure that clean blocks are always available by limiting the number of dirty blocks in the cache to a `max_dirty_blocks` threshold value. Once the cache hits this threshold, we fall back to write-through caching.

The flush thread writes dirty blocks to storage in the background. It uses asynchronous IO to batch blocks, and writes them proactively so that write requests avoid hitting the `max_dirty_blocks` threshold. The dirty map always refers to the latest version of a block, so only the last version is flushed when a block has been written multiple times. After a block is flushed, it is moved from the dirty map to the clean map. The flush thread waits when the number of dirty blocks reaches a low threshold value, unless it is signaled by the write-back flush policy to flush all dirty blocks (not shown in Table 3).

The write-back flush and write-back persist policies are implemented on a barrier request. The flush policy writes the dirty blocks to storage and waits for them to be durable by issuing a barrier request to storage. The persist policy makes the current blocks on storage durable and persists the dirty map on the flash device, performing the two operations concurrently and atomically.

These policies share much of the caching functionality. Next, we describe key differences in the mapping and allocation operations for the two policies.

4.2 Mapping Information

The mapping information allows translating the storage block numbers in IO requests to the blocks numbers for the cached blocks on flash. We store this mapping information in a BTree structure in memory because it enables fast lookup, and it can be persisted efficiently on flash.

4.2.1 Write-back Flush

The mapping information is kept solely in client memory for the write-back flush policy, because the cache contents are not needed after a client failure, as explained in Section 3.2.1. On a client restart, the flash cache is empty and the mapping information is repopulated on IO requests. Cache warming can help reduce this impact [23].

Read Request	Write Request

entry = find_mapping(bnr)
if (entry):
  # cache hit
  return read(flash, entry->flash_bnr)
else:
  # cache miss
  data = read(storage, bnr)
  if (flash_is_full):
    evict_clean_block()
  flash_bnr = alloc_block()
  write(flash, flash_bnr, data)
  insert_mapping(clean_map, bnr, flash_bnr)
  return data
	entry = find_mapping(bnr)
	if (entry):
	  free_block(entry->flash_bnr);
	  remove_mapping(entry->mapping, bnr);
	if (flash_is_full):
	  evict_clean_block()
	  if nr_dirty_blocks > max_dirty_blocks:
	    # fallback to write_through
	    write_through();
	    return
	flash_bnr = alloc_block()
	write(flash, flash_bnr, data)
	insert_mapping(dirty_map, bnr, flash_bnr)

Flush Thread	Barrier Request

foreach entry in dirty_map:
  # read dirty block from flash
  # and write to storage
  data = read(flash, entry->flash_bnr)
  write(storage, bnr, data)
  # move dirty block to clean state
  remove_mapping(dirty_map, bnr)
  insert_mapping(clean_map, bnr,
                entry->flash_bnr)
	if (policy == FLUSH):
	  signal(flush_thread)
	  wait(all_dirty_blocks_flushed)
	  barrier()
	else if (policy == PERSIST):
	  barrier()
	  persist_map(dirty_map, flash)

Table 3: IO Request Processing

4.2.2 Write-back Persist

The mapping information needs to be persisted atomically for the write-back persist policy, as explained in Section 3.2.2. This `persist_map` operation is performed on a barrier, as shown in Table 3. We implement atomic persist by using a copy-on-write BTree, similar to the approach used in the Btrfs file system [17].

Only the dirty mappings need to be persisted to ensure consistency and durability, since the clean blocks are already safe and can be retrieved from storage following a client restart. This option reduces barrier latency because the clean mappings do not have to be persisted. However, persisting all mappings may be beneficial because a client can restart with a warm cache.

We have chosen to persist only the dirty mapping information. To do so, we keep two separate BTrees, called the clean map and the dirty map, for the clean and dirty mappings. The dirty map is persisted on a barrier request; we call this the *persisted dirty map*. Compared to persisting all mappings using a single map, this separation benefits both read-heavy and random write workloads. In both cases, the dirty map will remain relatively small compared to the single map, which would either be large due to many clean mappings, or would have dirty mappings spread across the map, requiring many blocks to be persisted. An additional benefit is that recovery, which needs to read the persisted dirty map, is sped up.

When the flush thread writes a dirty block to storage, we move its mapping from the dirty map to the clean map, as shown in Table 3. This in-memory operation updates the dirty map, which is persisted at the next barrier.

4.3 Allocation Information

We use a bitmap to record block allocation information on the flash device. The bitmap indicates the blocks that are currently in use, either for the mapping metadata or the cached data blocks.

We do not persist the allocation bitmap to flash for several reasons. First, the bitmap does not need to be persisted at all for the write-back flush policy since the cache starts empty after client failure, as discussed in Section 4.2.1. Second, we separate the clean and dirty mapping information and only persist the dirty map for the write-back persist policy. As a result, we would also need to separate the clean and dirty allocation information and only persist the dirty allocation information to ensure consistency of the mapping and allocation information during recovery. Since we read in the dirty map during recovery anyway, which allows us to rebuild the allocation bitmap, this added complexity is not needed.

In the write-back persist policy, the cache blocks that are referenced in the persisted dirty map cannot be evicted even if they are clean, or else corruption may occur after recovery. For example, suppose that Block

A is updated and cached on flash Block F, and then the dirty map is persisted on a barrier. Now suppose Block B is updated and we evict Block A and overwrite flash Block F with the contents of Block B. On a failure, the dirty map would be reconstructed from the persisted dirty map, and so Block A would now map to Block F, which actually contains Block B. We solve this issue by maintaining a second in-memory bitmap, called the persist bitmap. This bitmap is updated on a barrier, and tracks the cache blocks in the persisted dirty map. A block is allocated only when it is free in both the allocation and the persist bitmaps, thus avoiding eviction of blocks referenced in the persisted dirty map.

4.4 Flushing Policies

Section 4.1 describes the basic operation of the flush thread. In this section, we describe two optimizations, flushing order and epoch-based flushing, that we have implemented for flushing dirty blocks.

4.4.1 Flushing Order

Our system supports flushing dirty blocks in two different orders, *LRU order* and *ascending order*. For the LRU order, the dirty blocks are maintained in a separate dirty LRU list. After the least-recently used dirty block is flushed, it is moved from the dirty LRU list to the clean LRU list. We use the last access timestamp to ensure that the flushed block is inserted in the clean LRU list in the correct order. As a result, after a cold dirty block is flushed, it is likely to be evicted soon.

We also support flushing dirty blocks in ascending order of storage block numbers. To do so, we use the dirty map (which stores its mappings in this sort order), as shown in the flush thread code in Table 3. Since the flash device can cache large amounts of data, we expect that flushing blocks in ascending order will significantly reduce seeks on the storage side compared to flushing blocks in LRU order. However, flushing in ascending order may have an affect on the cache hit rate because the flushed blocks may not be the least-recently used dirty blocks. As a result, warm clean blocks may be evicted before cold dirty blocks are flushed and evicted.

4.4.2 Epoch-Based Flushing

In the write-back flush policy, barrier request processing is a slow operation because all the dirty blocks need to be flushed to storage. Suppose an application thread has issued a barrier, e.g., by calling `fdatasync()`, but before the barrier finishes, another thread issues new writes. If barrier processing accepts these writes, it will take even longer to finish the barrier request, and with a high rate of incoming writes, barrier processing may never complete. Alternatively, new writes could be delayed until the completion of the barrier request. However, these writes may also incur high barrier latency, which defeats the goal of using a write-back policy to reduce write latencies.

We can take advantage of the barrier semantics, described in Section 3.1, to minimize delaying new writes, with *epoch-based flushing*. Each cache flush request starts a new epoch, and only the writes acknowledged within the epoch must become durable when the next flush request completes. This flushing of dirty blocks within an epoch requires two changes to the default write-back flush policy. First, the dirty mappings need to be split by epoch. Second, instead of waiting for all dirty blocks in the dirty map to be flushed (as shown in Table 3), the barrier request only waits until all the blocks associated with the dirty mappings in the current epoch are flushed. Since each barrier request starts a new epoch, and barrier processing can take time, multiple epochs may exist concurrently. To maintain data consistency, an epoch must be flushed before starting to flush the next epoch.

We maintain a separate BTree for all concurrent epochs in the dirty map. While epoch-based flushing increases concurrency because writes can be cached on the flash device while blocks are being flushed to storage on a barrier, it also increases the cost of the find and remove mapping operations because they need to search all BTrees. As a result, we have chosen a maximum limit of four concurrent epochs. If new writes are issued beyond this limit, then the writes are delayed.

5 Evaluation

To evaluate our caching policies, we have implemented a prototype caching system using the Linux device mapper framework. This framework enables the creation of virtual block devices that are transparent to the client, so minimal configuration is required to use the system.

Our implementation uses two Linux workqueues, serviced by two worker threads, for issuing asynchronous IO requests to the block layer. The first thread lies in the IO critical path and (i) issues read requests to the flash device on a cache hit or to storage on a cache miss, (ii) issues write requests to flash, and (iii) performs barrier processing, as shown in Table 3. The second thread is only used to issue write requests to flash to insert blocks into the cache following a read miss. We also use a flush thread to write dirty blocks to storage in the background. This thread issues read requests to flash, and write requests to storage. It uses asynchronous IO to batch requests, which helps hide network and storage latencies, thus improving flushing throughput.

Inspired by the journaled write-back policy [7], we implemented a variant of write-back flush called *write-back consistent* that flushes dirty data in each barrier epoch asynchronously. Similar to the flush policy, the consistent policy ensures that data in each epoch is flushed to storage before data in the next epoch. However, the consistent policy acknowledges the barrier operation immediately, without waiting for the flush operation, so it provides consistency but no durability on destructive failure.

We evaluate our write-back flush and persist caching policies by comparing them with four baseline policies, no caching (no flash used), write-through, write-back consistent and write-back caching. Of these, only the persist policy issues barrier requests to the flash device because it needs to persist its mapping atomically to the device. All the policies issue barriers to storage when the application makes a barrier request (see Table 3), with the exception of the write-back policy, which provides no reliability guarantees. Next, we present our experimental methodology and then the performance results.

5.1 Experimental Methodology

Our experimental setup consists of a storage server connected to a client machine with a flash device over 1 Gb Ethernet. The storage server runs Linux 3.11.2, has 4 Intel E7-4830 processors (32 cores in total), 256GB memory and a software RAID-6 volume consisting of 13 Hitachi HDS721010 SATA2 7200 RPM disks. Storage is served as an iSCSI target, using the in-kernel Linux-IO implementation. We disable the buffer cache on the server so that we can directly measure RAID performance, and also because our Linux-IO implementation ignores barriers when the buffer cache is enabled.

The client has 2 Xeon(R) E5-2650 processors and a 120GB Intel 510 Series SATA SSD. We use 8GB of the flash device as the client cache, with 2M mapping entries (1 per 4KB page). Each entry is 16 bytes, which together with the BTree structure, leads to a memory overhead of about 40MB for the mapping information.

We limit the memory on the client to 2GB so that our test data set will not completely fit in the client's buffer cache. In this setup, the ratio of the memory size and flash device capacity is similar to a mid-end, real-world storage server. For example, the NetApp FAS3270 storage system has 32GB RAM and a 100GB SSD when attached to a DS4243 disk shelf [14]. The client runs Linux 3.6.10 and mounts an Ext4 file system in ordered journal mode using the iSCSI target provided by the storage server. Ext4 initiates journal commits (leading to barriers issued to the block layer) every five seconds.

5.1.1 Workloads

We use Filebench 1.4.9.1 [5] to generate read-heavy, write-heavy and sync-heavy workloads on the client. For the read- and write-heavy workloads, barriers are initiated due to Ext4's commit interval. More frequent barriers occur due to application-level sync operations in the sync-heavy workload.

Read-heavy: webserver and webserver-large Webserver consists of several worker threads, each of which reads several whole files sequentially and then appends a small chunk of data to a log file. Files are selected using a uniform random distribution. Overall, webserver mostly performs sequential reads and some writes. We use two versions of this workload: webserver is a smaller version with 4GB of data, which can fit entirely on flash; webserver-large is a larger version, with 14GB of data, which causes cache capacity misses in our experiments.

Write-heavy: ms_nfs and ms_nfs-small ms_nfs is a metadata-heavy workload that emulates the behavior of a network file server. It includes a mix of file create, read, write and delete operations. The directory width, directory depth, and file size distributions in the data set are based on a recent metadata study by Microsoft Research [11]. Similar to webserver, we also use a compact version of ms_nfs, consisting of 6.5GB of data, while the original ms_nfs has 22GB.

Sync-heavy: varmail Varmail simulates a mail server application that issues frequent sync operations to ensure write ordering. For varmail, we use a single, default configuration with 4GB of data, which fits on flash, because a mail server typically does not have a large working set.

5.1.2 Metric

Filebench starts with a setup phase in which it populates a file system before running the workload. During setup, data is cached on flash and flushed in the background. We pause Filebench after setup finishes until the flush thread has stopped flushing data, to avoid interference between the setup phase and the workload. Then, we run each workload for 20 minutes.

Filebench reports the average IO operations/second (IOPS) for a workload at the end of the run. We modified it to report average IOPS every 30 seconds during the run. We found that this IOPS value varies in the first 10 minutes and then stabilizes, due to cache warming effects at both the buffer cache and the flash cache. We present steady-state IOPS results, by averaging the 20 IOPS readings taken in the last 10 minutes of the run.

5.2 Experimental Results

We first present the overall performance results for all the caching policies. We have enabled all flushing optimizations for our write-back policies. We flush in ascending order for both policies, and we use epoch-based flushing with 4 epochs for the write-back flush and write-back consistent policies. Finally, we show the impact of these flushing optimizations.

Figure 1 shows the average IOPS for the different caching policies for the three types of workload. As expected, all write caching policies perform comparably

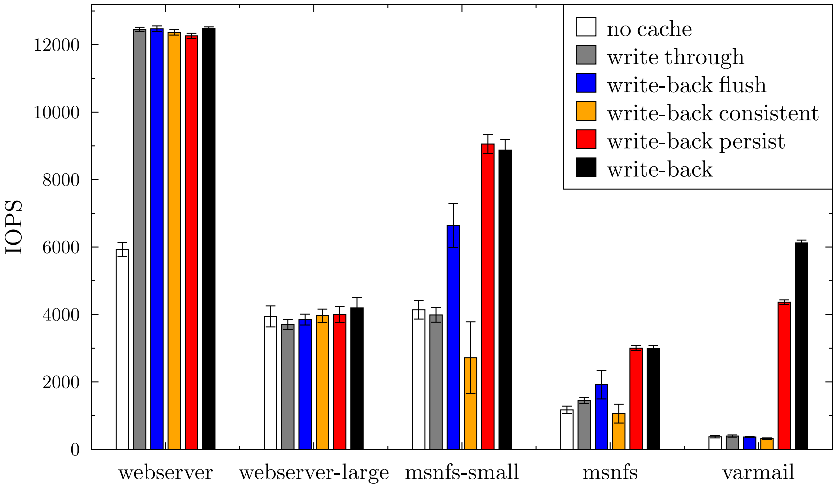


Figure 1: IOPS for different caching policies

for read-heavy workloads. For webserver, the caching policies increase IOPS by more than 2X compared to no-cache because the workload fits in the cache. We found that webserver saturates the flash bandwidth and hence write-back persist performs slightly worse because it needs to persist its mapping table to flash on barriers. In contrast, webserver-large has a low cache hit rate. Thus, it issues many reads to storage, making the storage a bottleneck, and so the no-cache policy performs as well as any caching policy.

The ms_nfs workloads create many dirty blocks and then free them, causing cache pollution and many cache misses, because the caching layer is unaware of freed blocks. Nonetheless, our write-back policies perform well, with write-back persist performing comparably to vanilla write-back because the workloads are write-heavy (84% and 58% write requests in ms_nfs-small and ms_nfs, respectively). With ms_nfs-small, write-through caching performs slightly worse than no-caching because the cache misses require filling the cache, however the difference is within the error bars. With the larger ms_nfs, storage becomes a bottleneck, and hence performance decreases for all workloads. However, this workload has a smaller ratio of write requests than ms_nfs-small and so the performance benefits of write-back caching over write-through caching are smaller.

In ms_nfs-small, the cache hit rate for write-back flush is 52%, while the hit rate for write-back persist is 79%, accounting for the difference in their performance. For write-back flush, the high barrier latency (due to flushing dirty blocks back to storage) causes filesystem journal commits to be delayed since the next transaction cannot commit until the previous one has completed. For ms_nfs-small, only 13 transactions were committed during the 20 minute run. This delay increases the epoch size (we observed a maximum of 4.6GB, with 2GB on average), which leads to dirty blocks occupying a large fraction of the flash cache. Read requests tend to be for clean blocks however, since recently written dirty blocks are more likely to be in the buffer cache, and the reduced number of clean blocks in the flash cache leads to a lower hit rate. We see a similar effect in the ms_nfs workload, although the hit rates are lower in both cases (45% for write-back flush vs. 60% for write-back persist).

Varmail is the most demanding workload for our policies. It issues `fsync()` calls frequently and waits for them to finish, making the workload sensitive to barrier latency. Write-back persist issues synchronous writes to flash and hence has significant overheads compared to the write-back policy. However, persist still performs much better than the other policies that issue synchronous writes to storage. The write-back flush policy performs worse than the write-through policy by 7%, because in our current implementation, the flush thread always performs an additional block read from flash to flush the block to storage.

Contrary to our expectation, the write-back consistent policy, which doesn't provide durability, performs worse than the write-back flush policy for the ms_nfs workloads. These workloads quickly consume all available epochs because each file-system commit issues a barrier that starts a new epoch, but the barriers themselves are not held up by flushing. When no epochs are available, all writes are blocked. We observed that epochs do not become large (as with write-back flush) but writes are frequently blocked due to having no available epochs. Increasing the number of epochs did not significantly improve performance. In contrast, the write-back flush policy delays barriers, but due to this delay it does not run out of epochs. This result suggests that the delay introduced by barrier-based flushing provides a natural control mechanism for avoiding other limits due to cache size, number of epochs, etc.

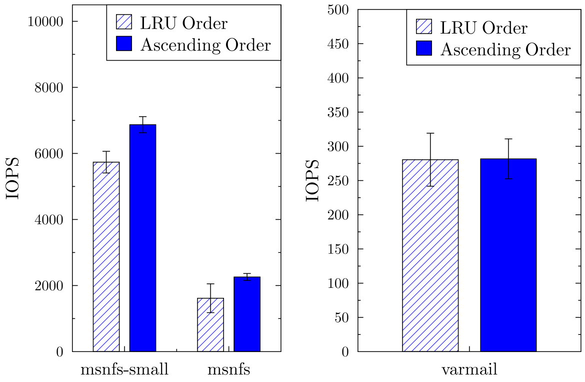


Figure 2: Effect of flushing order in the flush policy

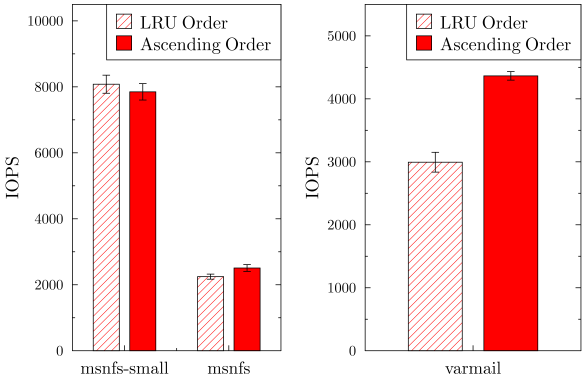


Figure 3: Effect of flushing order in the persist policy

5.2.1 Flushing Order Optimization

In this section, we evaluate the performance of flushing in two different orders, LRU order and ascending order, as described in Section 4.4.1. LRU order is expected to improve cache hit rate, while ascending order is expected to improve flushing throughput. For the read-heavy workloads, flushing is not a bottleneck and hence the flushing order does not affect performance.

Figure 2 shows the effect of flushing order for the write and sync-heavy workloads with the write-back flush policy. In this policy, the flushing operation can become a bottleneck because all the blocks dirtied in the last epoch need to be flushed on a barrier. Compared to LRU order, flushing in ascending order improves performance for the write-heavy workloads, by 19% for ms_nfs-small and 39% for ms_nfs. This result indicates that flushing is the bottleneck for write-heavy workloads and flushing in ascending order reduces disk seeks on storage. The varmail performance is not affected by the flushing order because there are few writes between barriers.

Figure 3 shows the effect of flushing order with the write-back persist policy. We measured the flushing throughput and found that flushing in ascending order performs significantly better than LRU order for both write and sync-heavy workloads. However, ms_nfs-small fits in the flash cache, making flash bandwidth the bottleneck, and hence the flushing order has minimal impact on performance. For ms_nfs, ascending order improves performance due to fewer seeks on storage.

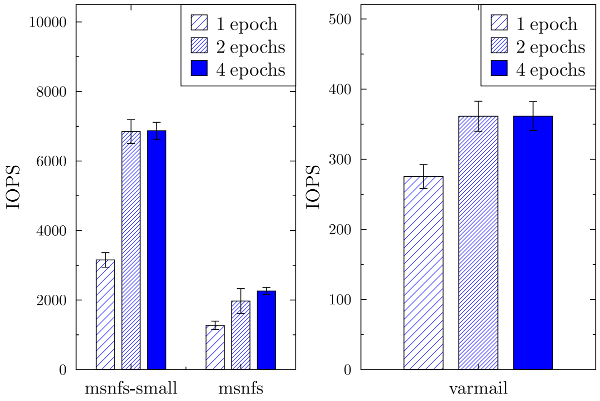


Figure 4: Effect of number of epochs in the flush policy

With varmail, flushing in ascending order helps migrate mapping entries from the dirty map to the clean map much more rapidly due to higher flushing throughput. Since varmail is very sensitive to barrier latency, and there are fewer entries in the dirty map to persist to flash at barriers, ascending order improves performance by 45% over LRU order.

We found that the hit rate did not change significantly when flushing in ascending order versus LRU order for write-back persist. We observed that the flushing operation is relatively efficient for these workloads, and as a result, cold blocks do not remain in the dirty LRU queue for long periods when flushing in ascending order. Once these blocks are flushed, they are moved to the clean queue, and then evicted. To assess the impact on the replacement policy, we logically combined the clean and the dirty mapping queues in timestamp order, and found that the tail of the clean queue is at most 8.5% from the tail of the combined queue in the worst case. As a result, ascending order flushing does not significantly affect the eviction decision and outperforms or is comparable to LRU flushing in all cases.

5.2.2 Epoch-Based Flushing Optimization

The write-back flush policy delays barrier requests because it needs to flush all dirty blocks back to storage. We implemented epoch-based flushing, described in Section 4.4.2, for the write-back flush policy to reduce this impact. We did not implement this optimization for the write-back persist policy because the dirty map can be persisted to the flash device efficiently on a barrier.

Figure 4 shows the benefits of using multiple epochs for flushing data. For both the write and sync heavy both workloads, using a maximum of two epochs improves performance significantly over using a single epoch. However, performance does not necessarily improve any further with four epochs because there may not be enough write concurrency in the workloads. Also, we keep a separate BTree for each epoch and need to search all the BTrees for the mapping operations, which may have a small impact on performance.

6 Related Work

There have been many recent proposals for using flash devices to improve IO performance, as we discuss here.

Koller et al. [7] present a client-side flash-based caching system that provides consistency on both recoverable and destructive failures. They present ordered write-back and journaled write-back policies, and their evaluation shows that journaled write-back performs better because it allows coalescing writes in an epoch. Unlike our write-back policies, both ordered and journaled write-back do not provide durability because they ignore barrier-related operations, such as `fsync()`. They also ignore disk cache flush commands, and thus do not guarantee consistency on storage failure.

Holland et al. [6] present a detailed trace-driven simulation study of client-side flash caching. They explore a large number of design decisions, including write policies in both the buffer and flash caches, unification of the two caches, cost of cache persistence, etc. They showed that write-back policies do not significantly improve performance. Write-through is sufficient because the caches are able to flush the dirty blocks to storage in time, and thus all application writes are asynchronous. However, their traces do not contain barrier requests (only reads and writes), thus they do not consider synchronous IO operations. Also, their simulation does not consider batching or reordering requests, which offers significant performance benefits, as we have shown.

Mercury [2] provides client-side flash caching that focuses on virtualization clusters in data centers. It uses the write-through policy for two reasons. First, their customers cannot handle losing any recent updates after a failure. Second, a virtual machine accessing storage can be migrated from one client machine to another at any time. With write-through caching, the caches are transparent to the migration mechanism. Mercury can persist cache metadata on flash, similar to the write-back persist policy. However, their motivation for persisting cache metadata is to reduce cache warm up times on a client reboot. Thus the cache metadata is only persisted on scheduled shutdown or reboot on the storage client.

FlashTier [19] presents an interface for caching on raw flash memory chips, rather than on flash storage with a flash translation layer (i.e., an SSD). Their approach benefits from integrating wear-level management and garbage collection with cache eviction, and using the out-of-band area on the raw flash chip to store the mapping tables. FlashTier complements our work because it allows using the flash device more efficiently.

Bcache [15] is a Linux-based caching system that supports caching on flash devices, similar to our system. It implements write-through caching and allows persisting metadata to flash. A comparison of our write-back policies against Bcache would be interesting. Bcache, however, does not support barrier requests in the kernel version that we used for our implementation, so the results would not be comparable.

Previous work on flash caching [6, 7] suggests that the flash cache hit rate, and thus the replacement policy, is crucial to performance. The Adaptive Replacement Cache (ARC) [10] algorithm is effective because it takes access frequency and recency into account, which makes ARC scan-resistant (i.e., it resists cache pollution on full table scans) and helps it adapt to the workload to improve cache hit rates. We have focused on the write-back policy and reliability, and have used only a simple LRU replacement policy.

Bonfire [23] shows that on-demand cache warm up (after system reboot) is slow because of growing flash caches. They warm the cache by monitoring I/O traces and loading hot data in bulk, which speeds up warm up time by 59% to 100% compared to on-demand warm up. We could use a similar approach for warming our cache.

7 Conclusions and Future Work

We have shown that a high-performance write-back caching system can support strong reliability guarantees. The key insight is that storage applications that require reliability already implement their own atomicity and durability schemes using write barriers, which provide the only reliable method for storing data durably on storage media. By leveraging these barriers, the caching system can provide both consistency and durability, and it can be implemented efficiently because applications have no reliability expectations between barriers.

We designed two flash-based caching policies called write-back flush and write-back persist. The write-back flush policy provides the same reliability guarantees as the write-through policy. The write-back persist policy assumes failure-free flash operation, and provides improved performance by flushing data to the flash cache rather than to storage on barrier requests, thereby reducing the latency of the barrier request.

Our evaluation showed three results. First, for read-heavy workloads, all caching policies, write-through or write-back, perform comparably. Second, our write-back policies provide higher performance than write-through caching for bursty and write-heavy workloads because IO requests can be batched and reordered. The dirty blocks in the flash cache can be batched and flushed asynchronously. They can also be reordered to improve the write access pattern and thus the flushing throughput on the storage side. Third, write-back persist performs comparably to write-back for all workloads, other than sync-heavy workloads, for which it still offers significant performance improvements over write-through caching.

In the future, we plan to use the trim command [3] to reduce cache pollution caused by freed, dirty blocks. We also plan to optimize the flush thread to avoid reading blocks from flash when they are available in the buffer cache. Finally, we plan to evaluate our write-back caching policies in virtualized environments [2].

References

[1] J. Axboe and S. Bhattacharya. http://lxr.free-electrons.com/source/Documentation/block/biodoc.txt#L826.

[2] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proc. of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[3] J. Corbet. Block layer discard requests. http://lwn.net/Articles/293658/.

[4] EMC. http://www.emc.com/collateral/hardware/data-sheet/h9581-vfcache-ds.pdf, 2013.

[5] Filebench. http://sourceforge.net/apps/mediawiki/filebench.

[6] D. A. Holland, E. Angelino, G. Wald, and M. I. Seltzer. Flash caching on the storage client. In *Proc. of the 2013 USENIX Annual Technical Conference*, ATC'13, pages 127–138, June 2013.

[7] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proc. of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 45–58, February 2013.

[8] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. Storage*, 6(3):13:1–13:26, Sept. 2010.

[9] C. Mason. ext3[34] barrier changes. http://lwn.net/Articles/283169/, 2008.

[10] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, pages 115–130, Mar. 2003.

[11] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–13, Feb. 2011.

[12] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: scaling down peak loads through I/O off-loading. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 15–28, Nov. 2008.

[13] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *Proc. of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 145–158, Mar. 2009.

[14] NetApp. http://www.netapp.com/us/products/storage-systems/fas3200/fas3200-tech-specs-compare.aspx.

[15] K. Overstreet. bcache. http://bcache.evilpiepirate.org/, 2013.

[16] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-system-based asynchronous mirroring for disaster recovery. In *Proc. of the First USENIX Conference on File and Storage Technologies (FAST)*, pages 117–129, Jan. 2002.

[17] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[18] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the USENIX Annual Technical Conference*, pages 1–14, June 2000.

[19] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A lightweight, consistent and durable storage cache. In *Proc. of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 267–280, Apr. 2012.

[20] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. In *Proc. of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pages 29–43, Feb. 2011.

[21] Tagged command queuing. Wikipedia. http://en.wikipedia.org/wiki/Tagged_Command_Queuing.

[22] Write barriers. Fedora Documentation. http://docs.fedoraproject.org/en-US/Fedora/14/html/Storage_Administration_Guide/writebarr.html.

[23] Y. Zhang, G. Soundararajan, M. W. Storer, L. N. Bairavasundaram, S. Subbiah, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Warming up storage-level caches with Bonfire. In *Proc. of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pages 59–72, Feb. 2013.

Flash on Rails: Consistent Flash Performance through Redundancy

Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, Scott Brandt
University of California, Santa Cruz
{*skourtis, optas, jayhawk, carlosm, scott*}*@cs.ucsc.edu*

Abstract

Modern applications and virtualization require fast and predictable storage. Hard-drives have low and unpredictable performance, while keeping everything in DRAM is still prohibitively expensive or unnecessary in many cases. Solid-state drives offer a balance between performance and cost and are becoming increasingly popular in storage systems, playing the role of large caches and permanent storage. Although their read performance is high and predictable, SSDs frequently block in the presence of writes, exceeding hard-drive latency and leading to unpredictable performance.

Many systems with mixed workloads have low latency requirements or require predictable performance and guarantees. In such cases the performance variance of SSDs becomes a problem for both predictability and raw performance. In this paper, we propose Rails, a design based on redundancy, which provides predictable performance and low latency for reads under read/write workloads by physically separating reads from writes. More specifically, reads achieve read-only performance while writes perform at least as well as before. We evaluate our design using micro-benchmarks and real traces, illustrating the performance benefits of Rails and read/write separation in solid-state drives.

1 Introduction

Virtualization and many other applications such as online analytics and transaction processing often require access to predictable, low-latency storage. Cost-effectively satisfying such performance requirements is hard due to the low and unpredictable performance of hard-drives, while storing all data in DRAM, in many cases, is still prohibitively expensive and often unnecessary. In addition, offering high performance storage in a virtualized cloud environment is more challenging due to the loss of predictability, throughput, and latency incurred by mixed workloads in a shared storage system. Given the popularity of cloud systems and virtualization, and the storage performance demands of modern applications, there is a clear need for scalable storage systems that provide high and predictable performance efficiently, under any mixture of workloads.

Solid-state drives and more generally flash memory have become an important component of many enterprise storage systems towards the goal of improving performance and predictability. They are commonly used as large caches and as permanent storage, often on top of hard-drives operating as long-term storage. A main advantage over hard-drives is their fast random access. One would like SSDs to be the answer to predictability, throughput, latency, and performance isolation for consolidated storage in cloud environments. Unfortunately, though, SSD performance is heavily workload dependent. Depending on the drive and the workload latencies as high as 100ms can occur frequently (for both writes and reads), making SSDs multiple times slower than hard-drives in such cases. In particular, we could only find a single SSD with predictable performance which, however, is multiple times more expensive than commodity drives, possibly due to additional hardware it employs (e.g., extra DRAM).

Such read-write interference results in unpredictable performance and creates significant challenges, especially in consolidated environments, where different types of workloads are mixed and clients require high throughput and low latency consistently, often in the form of reservations. Similar behavior has been observed in previous work [5, 6, 14, 17] for various device models and is well-known in the industry. Even so, most SSDs continue to exhibit unpredictability.

Although there is a continuing spread of solid-state drives in storage systems, research on providing efficient and predictable performance for SSDs is limited. In particular, most related work focuses on performance characteristics [5, 6, 4], while other work, including

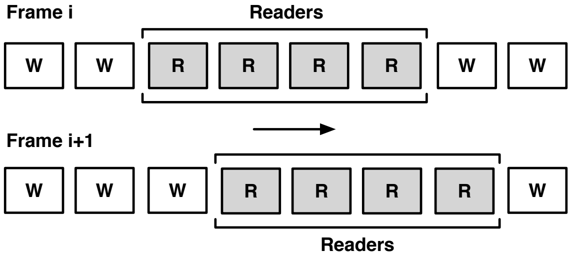


Figure 1: The sliding window moves along the drives. Drives inside the sliding window only perform reads and temporarily store writes in memory.

[1, 3, 8] is related to topics on the design of drives, such as wear-leveling, parallelism and the Flash Translation Layer (FTL). Instead, we use such performance observations to provide consistent performance. With regards to scheduling, FIOS [17] provides fair-sharing while trying to improve the drive efficiency. To mitigate performance variance, current flash-based solutions for the enterprise are often aggressively over provisioned, costing many times more than commodity solid-state drives, or offer lower write throughput. Given the fast spread of SSDs, we believe that providing predictable performance and low latency efficiently is important for many systems.

In this paper we propose and evaluate a method for achieving consistent performance and low latency under arbitrary read/write workloads by exploiting redundancy. Specifically, using our method read requests experience read-only throughput and latency, while write requests experience performance at least as well as before. To achieve this we physically separate reads from writes by placing the drives on a ring and using redundancy, e.g., replication. On this ring consider a sliding window (Figure 1), whose size depends on the desired data redundancy and read-to-write throughput ratio. The window moves along the ring one location at a time at a constant speed, transitioning between successive locations "instantaneously". Drives inside the sliding window do not perform any writes, hence bringing read-latency to read-only levels. All write requests received while inside the window are stored in memory (local cache/DRAM) and optionally to a log, and are actually written to the drive while outside the window.

2 Overview

The contribution of this paper is a design based on redundancy that provides read-only performance for reads under arbitrary read/write workloads. In other words, we provide consistent performance and minimal latency for reads while performing at least as well for writes as before. In addition, as we will see, there is opportunity to improve the write throughput through batch writing, however, this is out of the scope of this paper. Instead, we focus on achieving predictable and efficient read performance under read/write workloads. We present our results in three parts. In Section 3, we study the performance of multiple SSD models. We observe that in most cases their performance can become significantly unpredictable and that instantaneous performance depends heavily on past history of the workload. As we coarsen the measurement granularity we see, as expected, that at some point the worst-case throughput increases and stabilizes. This point, though, is quite significant, in the order of multiple seconds. Note that we illustrate specific drive performance characteristics to motivate our design for Rails rather than present a thorough study of the performance of each drive. Based on the above, in Section 4 we present Rails. In particular, we first show how to provide read/write separation using two drives and replication. Then we generalize our design to SSD arrays performing replication or erasure coding. Finally, we evaluate our method using replication under microbenchmarks and real workload traces.

2.1 System Notes

As described in Section 4.3, the design of Rails supports both replication and erasure coding. To this date we have implemented a prototype of the described design under replication rather than erasure coding. We believe that a thorough study of Rails under erasure coding requires a more extensive evaluation and is left as future work.

For our experiments we perform direct I/O to bypass the OS cache and use Kernel AIO to asynchronously dispatch requests to the raw device. To make our results easier to interpret, we do not use a filesystem. Limited experiments on top of ext3 and ext4 suggest our method would work in those cases. Moreover, our experiments were performed with both our queue and NCQ (Native Command Queueing) depth set to 31. Other queue depths had similar effects to what is presented in [6], that is throughput increased with the queue size. Finally, the SATA connector used was of 3.0Gb/s. For all our experiments we used the following SSD models:

	Model	Capacity	Cache	Year
A	Intel X-25E	65GB	16MB	2008
B	Intel 510	250GB	128MB	2011
C	Intel DC3700	400GB	512MB	2012
D	Samsung 840EVO	120GB	256MB	2013

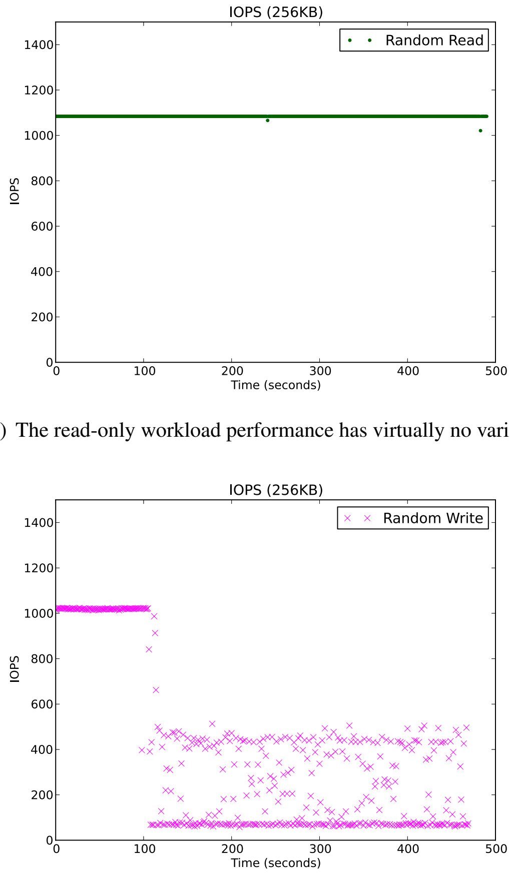
We chose the above drive models to develop a method, which unlike heuristics (Section 3.2), works under different types of drives, and especially commodity drives. A small number of recent data-center oriented models and in particular model *C* have placed a greater emphasis on performance stability. For the drives we are aware of, the stability either comes at a cost that is multiple times that of commodity drives (as with model *C*), or is achieved by lowering the random write throughput significantly compared to other models. In particular, commodity SSDs typically have a price between $0.5 and $1/GB while model *C* has a cost close to $2.5/GB. Most importantly, we are interested in a versatile, open solution supporting replication and erasure coding, which can be applied on existing systems by taking advantage of commodity hardware, rather than expensive black-box solutions.

3 Performance and Stability

Solid-state drives have orders-of-magnitude faster random access than hard-drives. On the other hand, SSDs are stateful and their performance depends heavily on the past history of the workload. The influence of writes on reads has been noted in prior work [5, 6, 14, 17] for various drive models, and is widely known in the industry. We first verify the behavior of reads and writes on drive *B*. By running a simple read workload with requests of 256KB over the first 200GB of drive *B*, we see from Figure 2(a) that the throughput is high and virtually variance-free. We noticed a similar behavior for smaller request sizes and for drive *A*. On the other hand, performing the same experiment but with random writes gives stable throughput up to a certain moment, after which the performance degrades and becomes unpredictable (Figure 2(b)), due to write-induced drive operations.

To illustrate the interference of writes on reads, we run the following two experiments on drive *B*. Consider two streams performing reads (one sequential and one random), and two streams performing writes (one sequential and one random). Each read stream has a dispatch rate of 0.4 while each write stream has a dispatch rate of 0.1, i.e., for each write we send four reads. In the first experiment, all streams perform operations on the same logical range of 100MB. Figure 3(a) shows the CDF of the throughput in IOPS (input/output per second) achieved by each stream over time. We note that the performance behavior is predictable. In the second experiment, we consider the same set of streams and dispatch rates, with the difference that the requests span the first 200GB, instead of only 100MB. Figure 3(b) illustrates the drive performance unpredictability under such conditions. We attribute this to the garbage collector not being able to keep up, which turns background operations into blocking ones.

Different SSD models can exhibit different throughput variance for the same workload. Performing random writes over the first 50GB of drive *A*, which has a capacity of 65GB, initially gives a throughput variance close to that of reads (figure skipped). Still, the average performance eventually degrades to that of *B*, with the total blocking time corresponding to more than 60% of the device time (Figure 4). Finally, although there is ongoing work, even newly released commodity SSDs (e.g., drive *D*) can have write latency that is 10 times that of reads, in addition to the significant write variance (figure skipped).

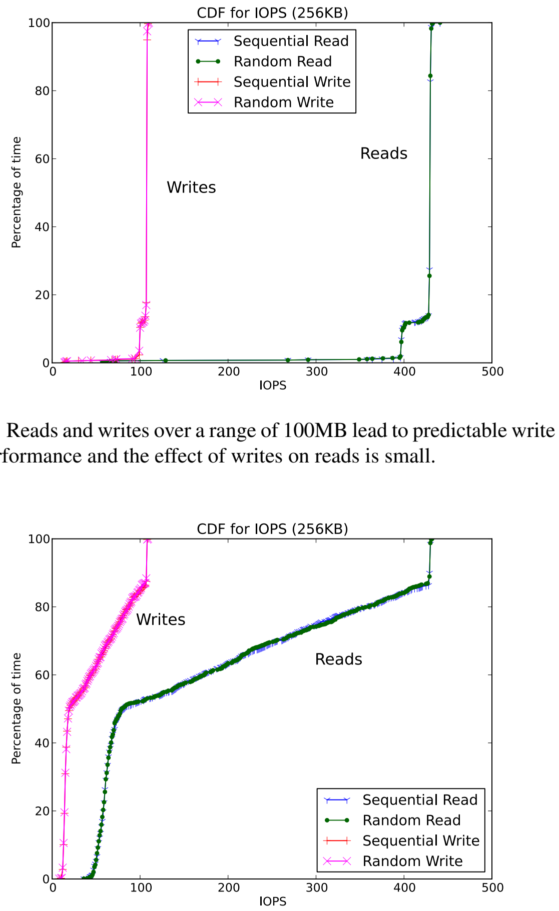


(a) The read-only workload performance has virtually no variance.

(b) When the drive has limited free space, random writes trigger the garbage collector resulting in unpredictable performance.

Figure 2: Under random writes the performance eventually drops and becomes unpredictable. (Drive B; 256KB)

Throughout our experiments we found that model *C* was the only drive with high and consistent performance under mixed read/write workloads. More specifically, after filling the drive multiple times by performing random writes, we presented it with a workload having a decreasing rate of (random) writes, from 90% down to 10%. To stress the device, we performed those writes over the whole drive's logical space. From Figure 5, we see that the read performance remains relatively predictable throughout the whole experiment. As mentioned earlier, a disadvantage of this device is its cost, part of which could be attributed to extra components it employs to



(a) Reads and writes over a range of 100MB lead to predictable write performance and the effect of writes on reads is small.

(b) Performing writes over 200GB leads to unpredictable read throughput due to write-induced blocking events.

Figure 3: The performance varies according to the writing range. (Drive B; 256KB)

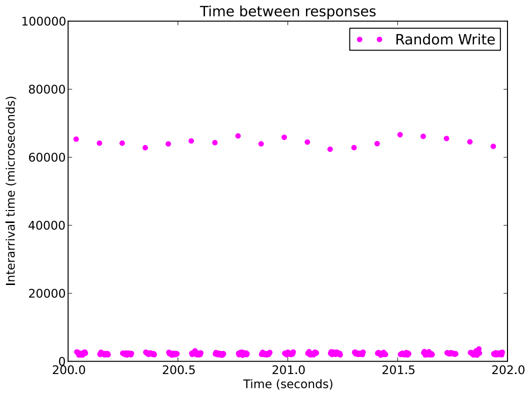


Figure 4: The drive blocks for over 600*ms*/sec, leading to high latencies for all queued requests. (Drive A; 256KB)

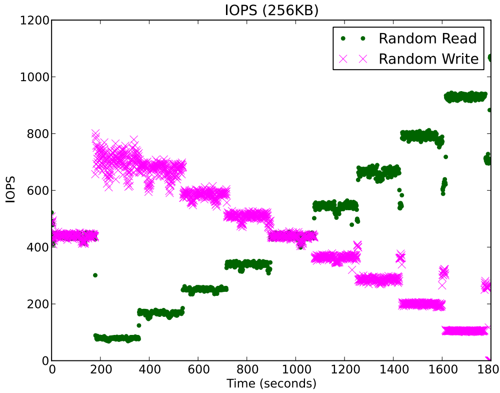


Figure 5: Random reads/writes at a decreasing write rate. Writes have little effect on reads. (Drive *C*; 256KB)

achieve this level of performance. As mentioned earlier, we are interested in an open, extensible and cheaper solution using existing commodity hardware. In addition, Rails, unlike model *C*, requires an amount of DRAM that is not proportional to the drive capacity. Finally, although we only tested drive *C* with a 3Gb/s SATA controller, we expect it to provide comparably predictable performance under 6Gb/s.

3.1 Performance over long periods

As mentioned in Section 1, writes targeting drives in read mode are accumulated and performed only when the drives start writing, which may be after multiple seconds. To that end, we are interested in the write throughput and predictability of drives over various time periods. For certain workloads and depending on the drive, the worst-case throughput can reach zero. A simple example of such workload on drive *A* consists of 4KB sequential writes. We noted that when the sequential writes enter a randomly written area, the throughput oscillates between 0 and 40,000 writes/sec. To illustrate how the throughput variance depends on the length of observation period, we computed the achieved throughput over different averaging window sizes, and for each window size we computed the corresponding CDF of throughputs. In Figure 6, we plot the 5%-ile of these throughput measurements as we increase the window size. We see that increasing the window size to about 5 seconds improves the 5%-ile, and that the increase is fast but then flattens out. That is, the throughput of SSDs over a window size of a few seconds becomes as predictable as the throughput over large window sizes. Drive *B* exhibits similar behavior. Finally, we found that the SSD write cache contributes to the throughput variance and although in our experiments we keep it enabled by default, disabling it improves stability but often at the cost of lower throughput, especially for small writes.

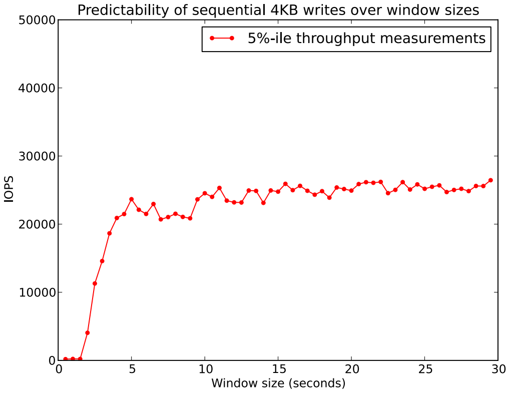


Figure 6: The bottom 5% throughput against the averaging window size. (Drive A; 4KB)

3.2 Heuristic Improvements

By studying drive models *A* and *B*, we found the behavior of *A*, which has a small cache, to be more easily affected by the workload type. First, writing sequentially over blocks that were previously written with a random pattern has low and unstable behavior, while writing sequentially over sequentially written blocks has high and stable performance. Although such patterns may appear under certain workloads and could be a filesystem optimization for certain drives, we cannot assume that in general. Moreover, switching from random to sequential writes on drive *A*, adds significant variance.

To reduce that variance we tried to disaggregate sequential from random writes (e.g., in 10-second batches). Doing so doubled the throughput and reduced the variance significantly (to 10% of the average). On the other hand, we should emphasize that the above heuristic does not improve the read variance of drive *B* unless the random writes happen over a small range. This strengthens the position of not relying on heuristics due to differences between SSDs. In contrast to the above, we next present a generic method for achieving efficient and predictable read performance under mixed workloads that is virtually independent of drive model and workload history.

4 Efficient and Predictable Performance

In the previous section, we observed that high latency events become common under read/write workloads leading to unpredictable performance, which is prohibitive for many applications. We now present a generic design based on redundancy that when applied on SSDs provides predictable performance and low latency for reads, by physically isolating them from writes. We expect this design to be significantly less prone to differences between drives than heuristics, and demonstrate its benefits under models *A* and *B*. In what follows, we first present a minimal version of our design, where we have two drives and perform replication. In Section 4.3, we generalize that to support more than two drives, erasure codes, and describe its achievable throughput.

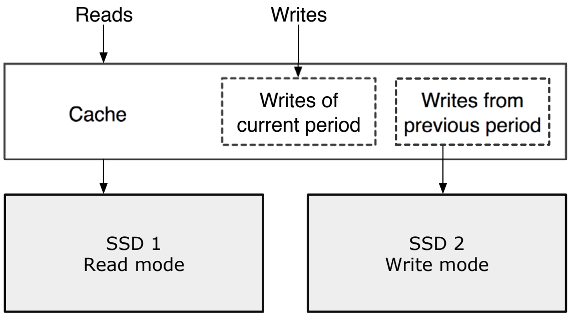


Figure 7: At any given time each of the two drives is either performing reads or writes. While one drive is reading the other drive is performing the writes of the previous period.

4.1 Basic Design

Solid-state drives have fast random access and can exhibit high performance. However, as shown in Section 3, depending on the current and past workloads, performance can degrade quickly. For example, performing random writes over a wide logical range of a drive can lead to high latencies for all queued requests due to write-induced blocking events (Figure 2(b)). Such events can last up to 100ms and account for a significant proportion of the device's time, e.g., 60% (Figure 4). Therefore, when mixing read and write workloads, reads also block considerably, which can be prohibitive.

We want a solution that provides read-only performance for reads under mixed workloads. SSD models differ from each other and a heuristic solution working on one model may not work well on another (Section 3.2). We are interested in an approach that works across various models. We propose a new design based on redundancy that achieves those goals by physically isolating reads from writes. By doing so, we nearly eliminate the latency that reads have to pay due to writes, which is crucial for many low-latency applications, such as online analytics. Moreover, we have the opportunity to further optimize reads and writes separately. Note that using a single drive and dispatching reads and writes in small time-based batches and prioritizing reads as in [17], may improve the performance under certain workloads and SSDs. However, it cannot eliminate the frequent blocking due to garbage collection under a generic workload.

The basic design, illustrated in Figure 7, works as follows: given two drives D_1 and D_2, we separate reads from writes by sending reads to D_1 and writes to D_2. After a variable amount of time $T \geq T_{min}$, the drives switch roles with D_1 performing writes and D_2 reads. When the switch takes place, D_1 performs all the writes D_2 completed that D_1 has not, so that the drives are in sync. We call those two consecutive time windows a *period*. If D_1 completes syncing and the window is not yet over ($t < T_{min}$), D_1 continues with new writes until $t \geq T_{min}$. In order to achieve the above, we place a cache on top of the drives. While the writing drive D_w performs the old writes all new writes are written to the cache. In terms of the write performance, by default, the user perceives write performance as perfectly stable and half that of a drive dedicated to writes. As will be discussed in Section 4.2.3, the above may be modified to allow new writes to be performed directly on the write drive, in addition to the cache, leading to a smaller memory footprint. In what follows we present certain properties of the above design and in Section 4.3 a generalization supporting an arbitrary number of drives, allowing us to trade read and write throughput, as well as erasure codes.

4.2 Properties and Challenges

4.2.1 Data consistency & fault-tolerance

All data is always accessible. In particular, by the above design, reads always have access to the latest data, possibly through the cache, independently of which drive is in read mode. This is because the union of the cache with any of the two drives always contains exactly the same (and latest) data. By the same argument, if any of the two drives fail at any point in time, there is no data loss and we continue having access to the latest data. While the system operates with one drive, the performance will be degraded until the replacement drive syncs up.

4.2.2 Cache size

Assuming we switch drive modes every T seconds and the write throughput of each drive is w MB/s, the cache size has to be at least $2T \times w$. This is because a total of $T \times w$ new writes are accepted while performing the previous $T \times w$ writes to each drive. We may lower that value to an average of $3/2 \times T \times w$ by removing from memory a write that is performed to both drives. As an example, if we switch every 10 seconds and the write throughput per drive is 200MB/s, then we need a cache of $T \times 2w = 4000$MB.

The above requires that the drives have the same throughput on average (over T seconds), which is reasonable to assume if T is not small (Figure 6). In general though, the write throughput of an SSD can vary in time depending on past workloads. This implies that even drives of the same model may not always have identical performance, It follows that a drive in our system may not manage to flush all its data while in write mode.

In a typical storage system an array of replica drives has the performance of its slowest drive. We want to retain that property while providing read/write separation to prevent the accumulated data of each drive from growing unbounded. To that end we ensure that the system accepts writes at the rate of its slowest drive through throttling. In particular, in the above 2-drive design, we ensure that the rate at which writes are accepted is $w/2$, i.e., half the write throughput of a drive. That condition is necessary to hold over large periods since replication implies that the write throughput, as perceived by the client, has to be half the drive throughput. Of course, extra cache may be added to handle bursts. Finally, the cache factor can become $w \times T$ by sacrificing up to half the read throughput if the syncing drive retrieves the required data from D_r instead of the cache. However, that would also sacrifice fault-tolerance and given the low cost of memory it may be an inferior choice.

4.2.3 Power failure

In an event of a power failure our design as described so far will result in a data loss of $T \times w$ MBs, which is less than 2GB in the above example. Shorter switching periods have a smaller possible data loss. Given the advantages of the original design, limited data loss may be tolerable by certain applications, such as applications streaming data that is not sensitive to small, bounded losses. However, other applications may not tolerate a potential data loss. To prevent data loss, non-volatile memory can be used to keep the design and implementation simple while retaining the option of predictable write throughput. As NVRAM becomes more popular and easily available, we expect that future implementations of Rails will assume its availability in the system.

In the absence of NVRAM, an approach to solve the power-failure problem is to turn incoming writes into synchronous ones. Assuming we split the bandwidth fairly between cached and incoming writes, the incoming $T \times w/2$ MBs of data is then written to D_w in addition to the cache. In that case, the amount of cache required reduces to an average of $T \times w/2$. In the above approach we first perform the writes of the previous period, and then any incoming writes. In practice, to avoid write starvation for the incoming writes, we can coalesce writes while sharing the bandwidth between the writes of the previous period and the incoming ones. As in write-back caches, and especially large flash caches, a challenge with the above is preserving write-ordering. Preserving the write order is required by a proportion of writes in certain applications (e.g., most file systems). To achieve this efficiently, a method such as ordered write-back [13] may be used, which preserves the original order during eviction by using dependency graphs. On the other hand, not all applications require write order preservation, including NoFS [7]. Note that other approaches such as performing writes to a permanent journal may be possible, especially in distributed storage systems where a separate journal drive often exists on each node. Finally, after the power is restored, we need to know which drive has the latest data, which can be achieved by storing metadata on D_w. The implementation details of the above are out of the scope of this paper.

4.2.4 Capacity and cost

Doubling the capacity required to store the same amount of data appears as doubling the storage cost. However, there are reasons why this is not entirely true. First, cheaper SSDs may be used in our design because we are taking away responsibility from the SSD controller by not mixing reads and writes. In other words, any reasonable SSD has high and stable read-only performance, and stable average write performance over large time intervals (Figure 6). Second, in practice, significant over-provisioning is already present to handle the unpredictable performance of mixed workloads. Third, providing a drive with a write-only load for multiple seconds instead of interleaving reads and writes is expected to improve its lifetime. Finally, and most importantly, Rails can take advantage of the redundancy that local and distributed systems often employ for fault-tolerance. In such cases, the hardware is already available. In particular, we next generalize the above design to more than two drives and reduce the storage space penalty through erasure codes while providing read/write separation.

4.3 Design Generalization

We now describe the generalization of the previous design to support an arbitrary number of identical SSDs and reader-to-writer drive ratios through erasure coding.

Imagine that we want to build a fault-tolerant storage system by using N identical solid-state drives connected over the network to a single controller. We will model redundancy as follows. Each object O stored will occupy $q|O|$ space, for some $q > 1$. Having fixed q, the best we can hope in terms of fault-tolerance and load-balancing is that the $q|O|$ bits used to represent O are distributed (evenly) among the N drives in such a way that O can be reconstituted from any set of N/q drives. A natural way to achieve load-balancing is the following. To handle a write request for an object O, each of the N drives receives a write request of size $|O| \times q/N$. To handle a read request for an object O, each of N/q randomly selected drives receives a read request of size $|O| \times q/N$.

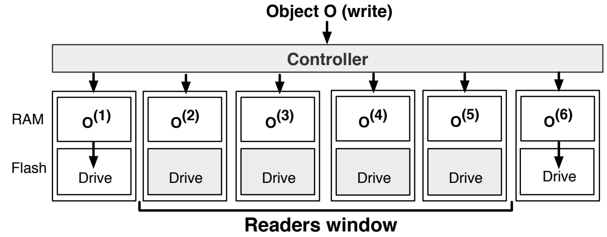


Figure 8: Each object is obfuscated and its chunks are spread across all drives. Reading drives store their chunk in memory until they become writers.

In the simple system above, writes are load-balanced deterministically since each write request places exactly the same load on each drive. Reads, on the other hand, are load-balanced via randomization. Each drive receives a stream of read and write requests whose interleaving mirrors the interleaving of read/write requests coming from the external world (more precisely, each external-world write request generates a write on each drive, while each external-world read request generates a read with probability $1/q$ on each drive.)

As discussed in Section 3, in the presence of read/write interleaving the write latency "pollutes" the variance of reads. We would like to avoid this latency contamination and bring read latency down to the levels that would be experienced if each drive was read-only. To this effect, we propose making the load-balancing of reads partially deterministic, as follows. Place the N drives on a ring. On this ring consider a sliding window of size s, such that $N/q \leq s \leq N$. The window moves along the ring one location at a time at a constant speed, transitioning between successive locations "instantaneously". The time it takes the window to complete a rotation is called the period P. The amount of time, P/N, that the window stays in each location is called a frame.

To handle a write request for an object O, each of the N drives receives one write request of size $|O| \times q/N$ (Figure 8, with $N = 6$ and $q = 3/2$). To handle a read request for an object O, out of the s drives in the window N/q drives are selected at random and each receives one read request of size $|O| \times q/N$. In other words, the only difference between the two systems is that reads are not handled by a random subset of nodes per read request, but by

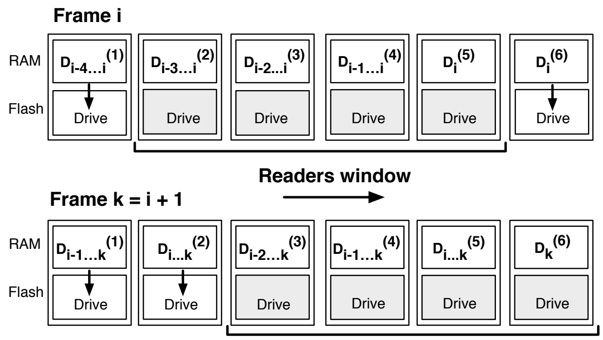


Figure 9: Each node m accumulates the incoming writes across frames $f \dots f'$ in memory, $D^{(m)}_{f \dots f'}$. While outside the reading window nodes flush their data.

random nodes from a coordinated subset which changes only after handling a large number of read requests.

In the new system, drives inside the sliding window do not perform any writes, hence bringing read-latency to read-only levels. Instead, while inside the window, each drive stores all write requests received in memory (local cache/DRAM) and optionally to a log. While outside the window, each drive empties all information in memory, i.e., it performs the actual writes (Figure 9). Thus, each drive is a read-only drive for $P/N \times s \geq P/q$ successive time units and a write-only drive for at most $P(1-1/q)$ successive time units.

Clearly, there is a tradeoff regarding P. The bigger P is, the longer the stretches for which each drive will only serve requests of one type and, therefore, the better the read performance predictability and latency. On the other hand, the smaller P is, the smaller the amount of memory needed for each drive.

4.3.1 Throughput Performance

Let us now look at the throughput difference between the two systems. The first system can accommodate any ratio of read and write loads, as long as the total demand placed on the system does not exceed capacity. Specifically, if r is the read-rate of each drive and w is the write-rate of each drive, then any load such that $R/r + Wq/w \leq N$ can be supported, where R and W are the read and write loads, respectively. In this system read and write workloads are mixed.

In the second system, s can be readily adjusted on the fly to any value in $[N/q, N]$, thus allowing the system to handle any read load up to the maximum possible rN. For each such choice of s, the other $N-s$ drives provide write throughput, which thus ranges between 0 and $W_{\text{sep}} = w \times (N - N/q)/q = w \times N(q-1)/q^2 \leq wN/4$. (Note that taking $s = 0$ creates a write-only system with optimal write-throughput wN/q.) We see that as long as the write load $W \leq W_{\text{sep}}$, by adjusting s the system performs perfect read/write separation and offers the read latency and predictability of a read-only system. We expect that in many shared storage systems, the reads-to-writes ratio and the redundancy are such that the above restriction is satisfied in the typical mode of operation. For example, for all $q \in [3/2, 3]$, having $R > 4W$ suffices.

When $W > W_{\text{sep}}$ some of the dedicated read nodes must become read/write nodes to handle the write load. As a result, read/write separation is only partial. Nevertheless, by construction, in every such case the second system performs at least as well as the first system in terms of read-latency. On the other hand, when performing replication ($q = N$) we have complete flexibility with respect to trading between read and write drives (or throughput) so we never have to mix reads and writes at a steady state.

Depending on the workload, ensuring that we adjust fast enough may require that drives switch modes quickly. In practice, to maintain low latency drives should not be changing mode quickly, otherwise reads could be blocked by write-induced background operations on the drive. Those operations could be triggered by previous writes. For example, we found that model B can switch between reads and writes every 5 seconds almost perfectly (Figure 10) while model A exhibits few blocking events that affect the system predictability (Figure 13(a)) when switching every 10 seconds.

We conclude that part of the performance predictability provided by Rails may have to be traded for the full utilization of every drive under fast-changing workloads. The strategy for managing that trade-off is out of the scope of this paper. Having said that, we expect that if the number of workloads seen by a shared storage is large enough, then the aggregate behavior will be stable enough and Rails would only see minor disruptions.

4.3.2 Feasibility and Efficiency

Consider the following four dimensions: storage space, reliability, computation and read performance variance. Systems performing replication have high space requirements. On the other hand, they offer reliability, and no computation costs for reconstruction. Moreover, applying our method improves their variance without affecting the other quantities. In other words, the system becomes strictly better.

Systems performing erasure coding have smaller storage space requirements and offer reliability, but add computation cost due to the reconstruction when there is a failure. Adding our method to such systems improves the performance variance. The price is that reading entails reconstruction, i.e., computation.

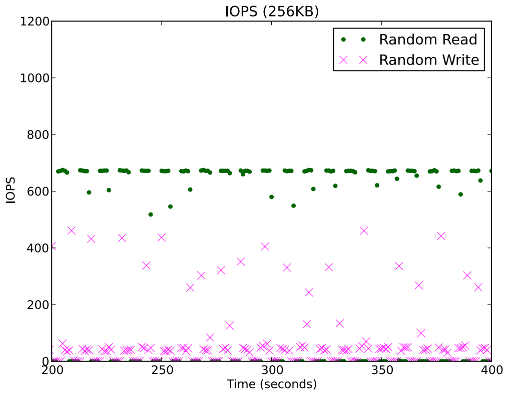
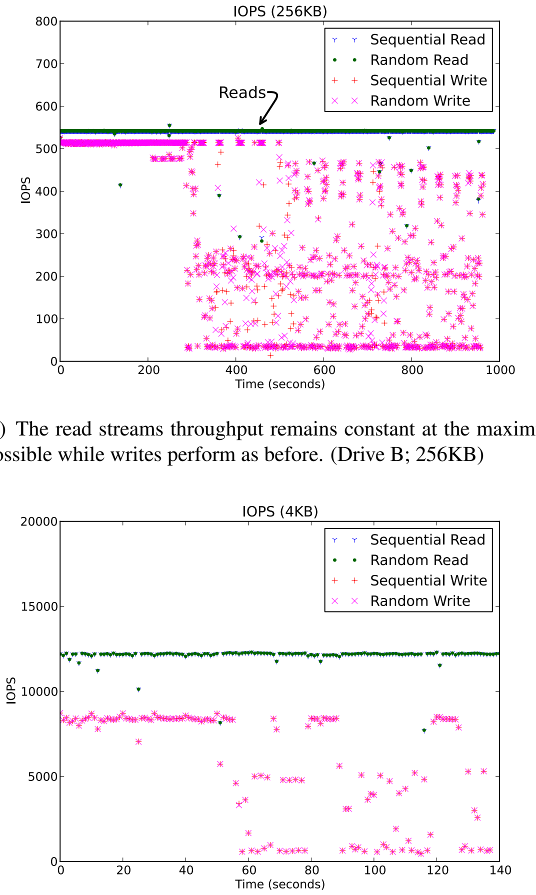


Figure 10: Switching modes as frequent as every 5 seconds creates little variance on reads. (Drive B; 256KB)



(a) The read streams throughput remains constant at the maximum possible while writes perform as before. (Drive B; 256KB)

(b) The read throughput remains stable at its maximum performance. (Drive B; 4KB)

Figure 11: Using Rails to physically separate reads from writes leads to a stable and high read performance.

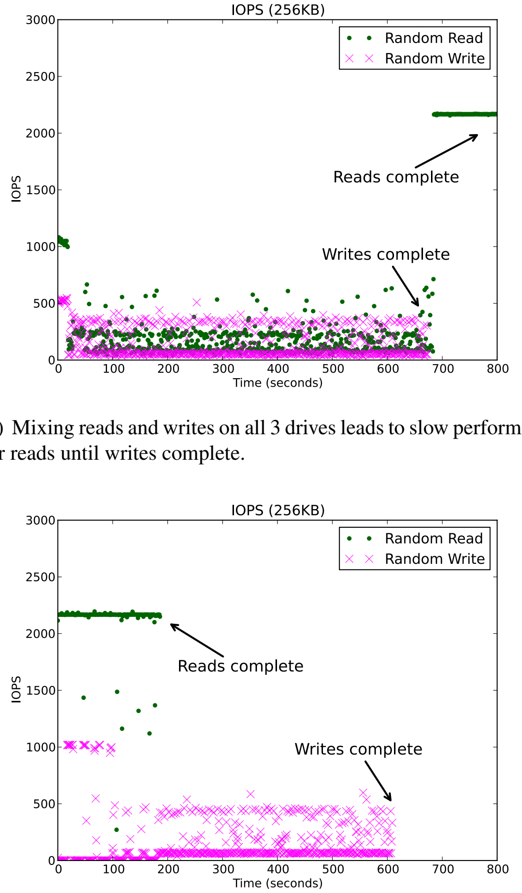
Nevertheless, reconstruction speed has improved [18] while optimizing degraded read performance is a possibility, as has been done for a single failure [12]. In practice, there are SSD systems performing reconstruction frequently to separate reads from writes, illustrating that reconstruction costs are tolerable for small N (e.g., 6). We are interested in the behavior of such systems under various codes as N grows, and identifying the value of N after which read/write separation becomes inefficient due to excessive computation or other bottlenecks.

4.4 Experimental Evaluation

We built a prototype of Rails as presented in Sections 4.1 and 4.3 using replication and verified that it provides predictable and efficient performance, and low latency for reads under read/write workloads using two and three drives. For simplicity, we ignored the possibility of cache hits or overwriting data still in the cache and focused on the worst-case performance. In what follows, we consider two drives that switch roles every $T_{min} = 10$ seconds. For the first experiment we used two instances of drive B. The workload consists of four streams, each sending requests of 256KB as fast as possible. From Figure 11(a), we see that reads happen at a total constant rate of 1100 reads/sec. and are not affected by writes. Writes however have a variable behavior as in earlier experiments, e.g., Figure 2(b). Without Rails reads have unstable performance due to the writes (Figure 3(b)).

Increasing the number of drives to three, and using the sliding window technique (Section 4.3), provides similar results. In particular, we set the number of read drives (window size) to two and therefore had a single write drive at a time. Figure 12(a) shows the performance without Rails is inconsistent when mixing reads and writes. In particular, in the first figure, the reads are being blocked by writes until writes complete. On the other hand, when using Rails the read performance is unaffected by the writes, and both the read and write workload finish earlier (Figure 12(b)). Note that when the reads complete, all three drives start performing writes.

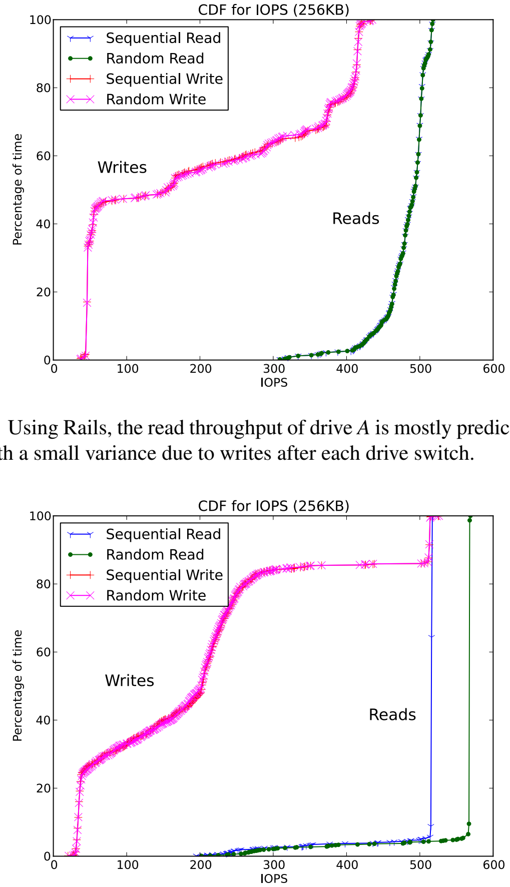
Although we physically separate reads from writes, in the worst-case there can still be interference due to remaining background work right after a window shift. In the previous experiment we noticed little interference, which was partly due to the drive itself. Specifically, from Figure 13(b) we see that reads have predictable performance around 95% of the time, which is significantly more predictable than without Rails (Figure 3(b)). Moreover, in Figure 13(a) we see that drive A has predictable read performance when using Rails, though reads do not appear as a perfect line, possibly due to its small cache. Since that may also happen with other drives we propose



(a) Mixing reads and writes on all 3 drives leads to slow performance for reads until writes complete.

(b) Using Rails to separate reads from writes across drives nearly eliminates the interference of writes on reads.

Figure 12: Client throughput under 3-replication, (a) without Rails, (b) with Rails. (Drive B; 256KB)



(a) Using Rails, the read throughput of drive *A* is mostly predictable, with a small variance due to writes after each drive switch.

(b) Using Rails, on drive *B* we can provide predictable performance for reads more than 95% of the time without any heuristics.

Figure 13: IOPS CDF using Rails on (a) drive *A*, (b) drive *B*. (256KB)

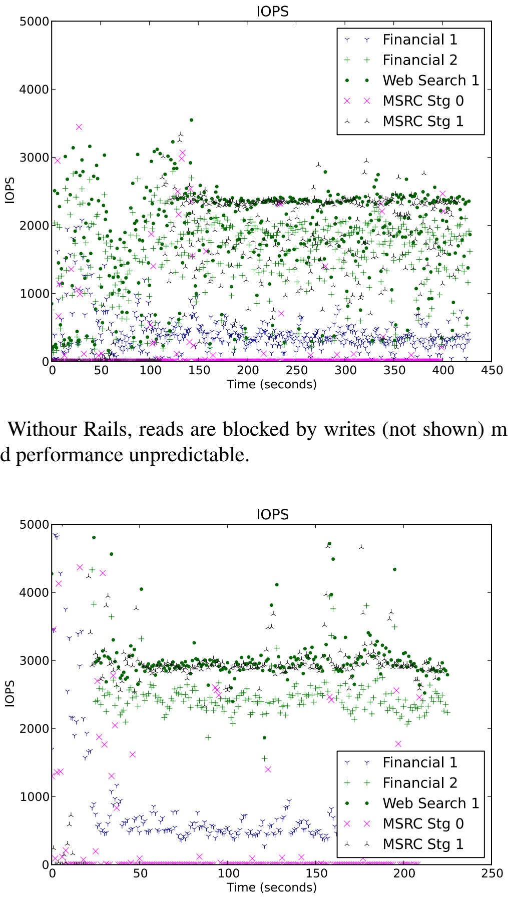
letting the write drive idle before each shift in order to reduce any remaining background work. That way, we found that the interference becomes minimal and 99% of the time the throughput is stable. If providing QoS, that idle time can be charged to the write streams, since they are responsible for the blocking. Small amounts of interference may be acceptable, however, certain users may prefer to sacrifice part of the write throughput to further reduce the chance of high read latency.

The random write throughput achieved by the commodity drives we used (models *A*, *B*, and *D*) drops significantly after some number of operations (Figure 2(b)). Instead, model *C*, which is more expensive as discussed earlier, retains its write performance. We believe there is an opportunity to increase the write throughput in Rails for commodity drives through batch writing, or through a version of SFS [14] adapted to redundancy. That is because many writes in Rails happen in the background.

4.5 Evaluation with Traces

We next evaluate Rails with two drives (of model *B*) and replication using the Stg dataset from the MSR Cambridge Traces [16], OLTP traces from a financial institution and traces from a popular search engine [26]. Other combinations of MSRC traces gave us similar results with respect to read/write isolation and skip them. For the following experiments we performed large writes to fill the drive cache before running the traces. Evaluating results using traces can be more challenging to interpret due to request size differences leading to a variable throughput even under a storage system capable of delivering perfectly predictable performance.

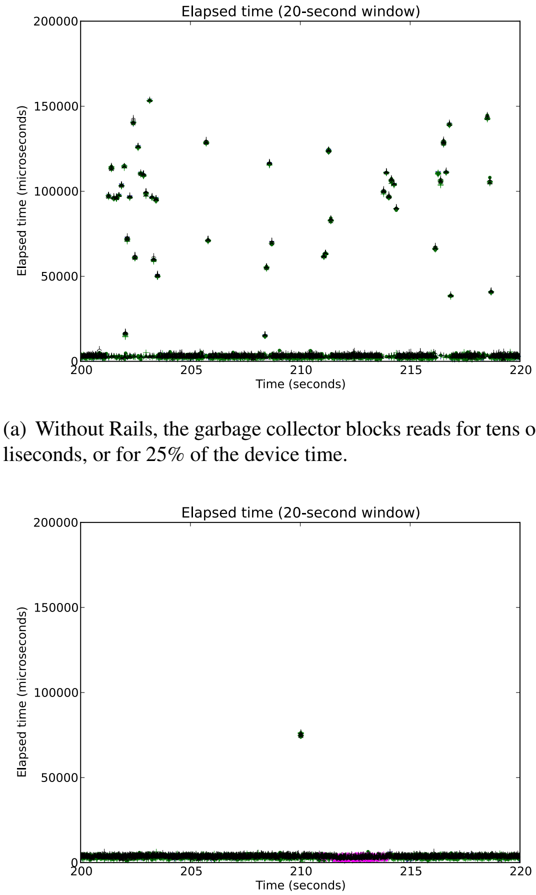
In terms of read/write isolation, Figure 14(a) shows the high variance of the read throughput when mixing reads and writes under a single device. The write plots for both cases are skipped as they are as unstable as Figure 14(a). Under the same workload our method pro-



(a) Withour Rails, reads are blocked by writes (not shown) making read performance unpredictable.

(b) With Rails, reads are not affected by writes (not shown).

Figure 14: Read throughput (a) without Rails, (b) with Rails, under a mixture of real workloads. (Drive B)



(a) Without Rails, the garbage collector blocks reads for tens of milliseconds, or for 25% of the device time.

(b) With Rails, reads are virtually unaffected by writes - they are blocked for less than %1 of the time.

Figure 15: High-latency events (a) without Rails, (b) with Rails, using traces of real workloads. (Drive B)

vides predictable performance (Figure 14(b)) in the sense that high-latency events become rare. To clearly see that, Figure 15(a) focuses on twenty arbitrary seconds of the same experiment and illustrates that without Rails there are multiple response times in the range of 100ms. Looking more closely, we see that about 25% of the time reads are blocked due to the write operations. On the other hand, from Figure 15(b) we see that Rails nearly eliminates the high latencies, therefore providing read-only response time that is low and predictable, almost always.

5 Related Work

Multiple papers study the performance characteristics of SSDs. uFlip [4] presents a benchmark and illustrates flash performance patterns, while the authors in [6] study the effect of parallelism on performance. The work in [5] includes a set of experiments on the effect of reads/writes and access patterns on performance. In addition, Rajimwale et al. present system-level assumptions that need to be revisited in the context of SSDs [20]. Other work focuses on design improvements, and touches on a number of aspects of performance such as parallelism and write ordering [1]. The authors in [8] propose a solution for write amplification, while [3] focuses on write endurance and its implications on disk scheduling. Moreover, Grupp et al. focus on the future of flash and the relation between its density and performance [9].

In the context of hard-drive storage, DCD [10] proposes adding a log drive to cache small writes and destage them to the data drive. Fahrrad [19] treats sequential and random requests differently to provide predictable performance, while QBox [24] takes a similar approach for black-box storage. Gecko [23] is a log-structured design for reducing workload interference in hard-drive storage. In particular, it spreads the log across multiple hard-drives, therefore decreasing the effect of garbage collection on the incoming workload.

There is little work taking advantage of performance results in the context of scheduling and QoS. A fair scheduler optimized for flash is FIOS [17] (and its successor FlashFQ [22]). Part of FIOS gives priority to reads over writes, which provides improvements for certain drive models. However, FIOS is designed as an efficient flash scheduler rather than a method for guaranteeing low latency. Instead, we use a drive-agnostic method to achieve minimal latency. In another direction, SFS [14] presents a filesystem designed to improve write performance by turning random writes to sequential ones. SSDs are often used as a high performance tier (a large cache) on top of hard-drives [2, 15]. Our solution may be simplified and applied in such cases. Finally, there is recent work on the applications of erasure coding on large-scale storage [11, 21]. We expect our method to be applicable in practice on storage systems using erasure coding to separate reads from writes.

6 Conclusion

The performance of SSDs degrades and becomes significantly unpredictable under demanding read/write workloads. In this paper, we introduced Rails, an approach based on redundancy that physically separates reads from writes to achieve read-only performance in the presence of writes. Through experiments, we demonstrated that under replication, Rails enables efficient and predictable performance for reads under read/write workloads. A direction for future work is studying the implementation details of Rails regarding write order preservation in the lack of NVRAM. Finally, we plan to study the scalability of Rails using erasure codes, as well as its application on peta-scale distributed flash-only storage systems, as proposed in [25].

Acknowledgements: We would like to thank the anonymous USENIX ATC reviewers and our shepherd Kai Shen for comments that helped improve this paper.

References

[1] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. Design tradeoffs for SSD performance. In *USENIX ATC'08* (2008).

[2] Albrecht, C., Merchant, A., et al. Janus: Optimal flash provisioning for cloud storage workloads. In *ATC '13* (2013).

[3] Boboila, S., and Desnoyers, P. Write endurance in flash drives: measurements and analysis. In *USENIX FAST'10* (2010).

[4] Bouganim, L., Jónsson, B., and Bonnet, P. uFLIP: Understanding flash IO patterns. In *CIDR '09* (2009), CIDR.

[5] Chen, F., Koufaty, D. A., and Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09* (2009), ACM.

[6] Chen, F., Lee, R., and Zhang, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *HPCA '11* (2011), IEEE.

[7] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency without ordering. In *USENIX FAST12* (2012).

[8] Desnoyers, P. Analytic modeling of SSD write performance. In *SYSTOR '12* (2012), ACM.

[9] Grupp, L. M., Davis, J. D., and Swanson, S. The bleak future of NAND flash memory. In *USENIX FAST'12* (2012).

[10] Hu, Y., and Yang, Q. DCD-Disk Caching Disk: A new approach for boosting I/O performance. In *ISCA '96* (1996), ACM.

[11] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure coding in windows azure storage. In *USENIX ATC'12* (2012).

[12] Khan, O., Burns, R., Plank, J., Pierce, W., and Huang, C. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *USENIX FAST'12* (2012).

[13] Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N., and Zhao, M. Write policies for host-side flash caches. In *USENIX FAST'13* (2013), vol. 13.

[14] Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y. I. SFS: Random write considered harmful in solid state drives. In *USENIX FAST'12* (2012), pp. 12–12.

[15] Moshayedi, M., and Wilkison, P. Enterprise SSDs. *Queue 6*, 4 (July 2008).

[16] Narayanan, D., Donnelly, A., and Rowstron, A. Write off-loading: practical power management for enterprise storage. In *USENIX FAST'08* (2008).

[17] Park, S., and Shen, K. FIOS: a fair, efficient flash I/O scheduler. In *USENIX FAST'12* (2012).

[18] Plank, J. S., Greenan, K. M., and Miller, E. L. Screaming fast galois field arithmetic using Intel SIMD instructions. In *USENIX FAST'13* (2013).

[19] Povzner, A., Kaldewey, T., Brandt, S., Golding, R., Wong, T. M., and Maltzahn, C. Efficient guaranteed disk request scheduling with Fahrrad. In *Eurosys '08* (2008), ACM.

[20] Rajimwale, A., Prabhakaran, V., and Davis, J. D. Block management in solid-state devices. In *USENIX ATC'09* (2009).

[21] Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A. G., et al. XORing elephants: novel erasure codes for big data. In *PVLDB'13* (2013), VLDB Endowment.

[22] Shen, K., and Park, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX ATC'13* (2013).

[23] Shin, J. Y., Balakrishnan, M., Marian, T., and Weatherspoon, H. Gecko: Contention-oblivious disk arrays for cloud storage. In *USENIX FAST'13* (2013).

[24] Skourtis, D., Kato, S., and Brandt, S. QBox: guaranteeing I/O performance on black box storage systems. In *HPDC '12* (2012), ACM.

[25] Skourtis, D., Watkins, N., Achlioptas, D., Maltzahn, C., and Brandt, S. Latency minimization in SSD clusters for free. Tech. Rep. UCSC-SOE-13-10, UC Santa Cruz, June 2013.

[26] OLTP traces from the University of Massachusetts Amherst trace repository. http://traces.cs.umass.edu/index.php/Storage.

I/O Speculation for the Microsecond Era

Michael Wei[†], Matias Bjørling[‡], Philippe Bonnet[‡], Steven Swanson[†]
[†]*University of California, San Diego* [‡]*IT University of Copenhagen*

Abstract

Microsecond latencies and access times will soon dominate most datacenter I/O workloads, thanks to improvements in both storage and networking technologies. Current techniques for dealing with I/O latency are targeted for either very fast (nanosecond) or slow (millisecond) devices. These techniques are suboptimal for microsecond devices - they either block the processor for tens of microseconds or yield the processor only to be ready again microseconds later. Speculation is an alternative technique that resolves the issues of yielding and blocking by enabling an application to continue running until the application produces an externally visible side effect. State-of-the-art techniques for speculating on I/O requests involve checkpointing, which can take up to a millisecond, squandering any of the performance benefits microsecond scale devices have to offer. In this paper, we survey how speculation can address the challenges that microsecond scale devices will bring. We measure applications for the potential benefit to be gained from speculation and examine several classes of speculation techniques. In addition, we propose two new techniques, hardware checkpoint and checkpoint-free speculation. Our exploration suggests that speculation will enable systems to extract the maximum performance of I/O devices in the microsecond era.

1 Introduction

We are at the dawn of the *microsecond era*: current state-of-the-art NAND-based Solid State Disks (SSDs) offer latencies in the sub-100μs range at reasonable cost [16, 14]. At the same time, improvements in network software and hardware have brought network latencies closer to their physical limits, enabling sub-100μs communication latencies. The net result of these developments is that the datacenter will soon be dominated by microsecond-scale I/O requests.

Device	Read	Write
Millisecond Scale		
10G Intercontinental RPC	100 ms	100 ms
10G Intracontinental RPC	20 ms	20 ms
Hard Disk	10 ms	10 ms
10G Interregional RPC	1 ms	1 ms
Microsecond Scale		
10G Intraregional RPC	300 μs	300 μs
SATA NAND SSD	200 μs	50 μs
PCIe/NVMe NAND SSD	60 μs	15 μs
10Ge Inter-Datacenter RPC	10 μs	10 μs
40Ge Inter-Datacenter RPC	5 μs	5 μs
PCM SSD	5 μs	5 μs
Nanosecond Scale		
40 Gb Intra-Rack RPC	100 ns	100 ns
DRAM	10 ns	10 ns
STT-RAM	<10 ns	<10 ns

Table 1: **I/O device latencies.** Typical random read and write latencies for a variety of I/O devices. The majority of I/Os in the datacenter will be in the microsecond range.

Today, an operating system uses one of two options when an application makes an I/O request: either it can block and poll for the I/O to complete, or it can complete the I/O asynchronously by placing the request in a queue and yielding the processor to another thread or application until the I/O completes. Polling is an effective strategy for devices with submicrosecond latency [2, 20], while programmers have used yielding and asynchronous I/O completion for decades on devices with millisecond latencies, such as disk. Neither of these strategies, however, is a perfect fit for microsecond-scale I/O requests: blocking will prevent the processor from doing work for tens of microseconds, while yielding may reduce performance by increasing the overhead of each I/O operation.

A third option exists as a solution for dispatching I/O requests, *speculation*. Under the speculation strategy, the operating system completes I/O operations speculatively, returning control to the application without yielding. The operating system monitors the application: in the case of a write operation, the operating system blocks the application if it makes a side-effect, and in the case of a read operation, the operating system blocks the application if it attempts to use data that the OS has not read yet. In addition, the operating system may have a mechanism to rollback if the I/O operation does not complete successfully. By speculating, an application can continue to do useful work even if the I/O has not completed. In the context of microsecond-scale I/O, speculation can be extremely valuable since, as we discuss in the next section, there is often enough work available to hide microsecond latencies. We expect that storage class memories, such as phase-change memory (PCM), will especially benefit from speculation since their access latencies are unpredictable and variable [12].

Any performance benefit to be gained from speculation is dependent upon the performance overhead of speculating. Previous work in I/O speculation [9, 10] has relied on checkpointing to enable rollback in case of write failure. Even lightweight checkpointing, which utilizes copy-on-write techniques, has a significant overhead which can exceed the access latency of microsecond devices.

In this paper, we survey speculation in the context of microsecond-scale I/O devices, and attempt to quantify the performance gains that speculation has to offer. We then explore several techniques for speculation, which includes exploring existing software-based checkpointing techniques. We also propose new techniques which exploit the semantics of the traditional I/O interface. We find that while speculation could allow us to maximize the performance of microsecond scale devices, current techniques for speculation cannot deliver the performance which microsecond scale devices require.

2 Background

Past research has shown that current systems have built-in the assumption that I/O is dominated by millisecond scale requests [17, 2]. These assumptions have impacted the core design of the applications and operating systems we use today, and may not be valid in a world where I/O is an order of magnitude faster. In this Section, we discuss the two major strategies for handling I/O and show that they do not adequately address the needs of microsecond-scale devices, and we give an overview of I/O speculation.

2.1 Interfaces versus Strategies

When an application issues a request for an I/O, it uses an *interface* to make that request. A common example is the POSIX `write/read` interface, where applications make I/O requests by issuing blocking calls. Another example is the POSIX asynchronous I/O interface, in which applications enqueue requests to complete asynchronously and retrieve the status of their completion at some later time.

Contrast interfaces with *strategies*, which refers to how the operating system actually handles I/O requests. For example, even though the `write` interface is blocking, the operating system may choose to handle the I/O asynchronously, yielding the processor to some other thread.

In this work, we primarily discuss operating system strategies for handling I/O requests, and assume that application developers are free to choose interfaces.

2.2 Asynchronous I/O - Yielding

Yielding, or the asynchronous I/O strategy, follows the traditional pattern for handling I/O requests within the operating system: when a userspace thread issues an I/O request, the I/O subsystem issues the request and the scheduler places the thread in an I/O wait state. Once the I/O device completes the request, it informs the operating system, usually by means of a hardware interrupt, and the operating system then places the thread into a ready state, which enables the thread to resume when it is rescheduled.

Yielding has the advantage of allowing other tasks to utilize the CPU while the I/O is being processed. However, yielding introduces significant overhead, which is particularly relevant for fast I/O devices [2, 20]. For example, yielding introduces contexts switches, cache and TLB pollution as well as interrupt overhead that may exceed the cost of doing the I/O itself. These overheads are typically in the microsecond range, which makes the cost of yielding minimal when dealing with millisecond latencies as with disks and slow WANs, but high when dealing with nanosecond devices, such as fast NVMs.

2.3 Synchronous I/O - Blocking

Blocking, or the synchronous I/O strategy, is a solution for dealing with devices like fast NVMs. Instead of yielding the CPU in order for I/O to complete, blocking prevents unnecessary context switches by having the application poll for I/O completions, keeping the entire context of execution within the executing thread. Typically, the application stays in a spin-wait loop until the I/O completes, and resumes execution once the device flags the I/O as complete.

Blocking prevents the CPU from incurring the cost of context switches, cache and TLB pollution as well as interrupt overhead that the yielding strategy incurs. However, the CPU is stalled for the amount of time the I/O takes to complete. If the I/O is fast, then this strategy is optimal since the amount of time spent waiting is much shorter than the amount of CPU time lost due to software overheads. However, if the I/O is in the milliseconds range, this strategy wastes many CPU cycles in the spin-wait loop.

2.4 Addressing Microsecond-Scale Devices

Microsecond-scale devices do not fit perfectly into either strategy: blocking may cause the processor to block for a significant amount of time, preventing useful work from being done, and yielding may introduce overheads that may not have been significant with millisecond-scale devices, but may exceed the time to access a microsecond scale device. Current literature [2, 20] typically recommends that devices with microsecond ($\geq 5\mu$s) latencies use the yielding strategy.

2.5 I/O Speculation

Speculation is a widely employed technique in which a execution occurs before it is known whether it is needed or correct. Most modern processors use speculation: for example, branch predictors resolve branches before the branch path has been calculated [15]. Optimistic concurrency control in database systems enables multiple transactions to proceed before conflicts are resolved [5]. Prefetching systems attempt to make data available before it is known to be needed [8]. In all these speculative systems, speculation has no effect on correctness – if a misspeculation occurs either it has no effect on correctness or the system can rollback state as if no speculation had occurred in the first place.

I/O requests are a good speculation candidate for several reasons. The results of an I/O request are simple and predictable. In the case of a write, the write either succeeds or fails. For a read, the request usually returns success or failure immediately, and a buffer with the requested data is filled. In the common case, I/Os typically succeed – failures such as a disk error or an unreachable host are usually exceptional conditions that do not occur in a typical application run.

We depict the basic process of speculation in Figure 1. In order to speculate, a speculative context is created first. Creating a speculative context incurs a performance penalty ($t_{speculate}$), but once the context is created, the task can speculatively execute for some time ($t_{spec\_execute}$), doing useful work until it is no longer safe to speculate (t_{wait}). In the meantime, the kernel can dispatch the I/O request asynchronously (t_{io_start}). Once the I/O request completes (t_{io_comp}), the kernel commits the speculative execution if the request is successful, or aborts the speculative execution if it is not.

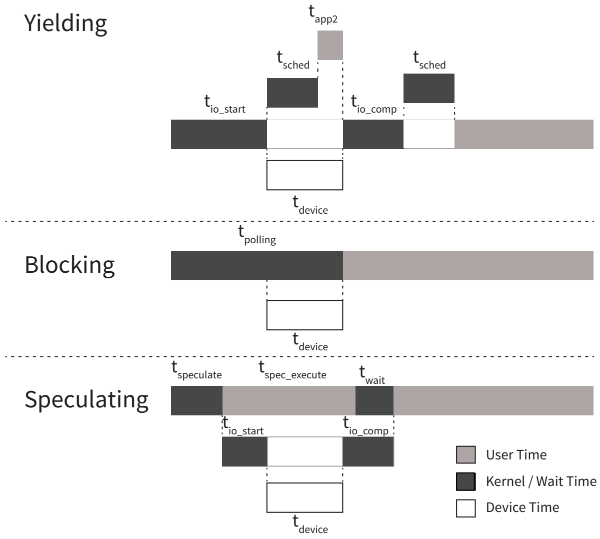


Figure 1: **Cost breakdown by strategy.** These diagrams show the relative costs for the yielding, blocking and speculating strategies.

In contrast to the blocking strategy, where the application cannot do useful work while the kernel is polling for the device to complete, speculation allows the application to perform useful work while the I/O is being dispatched. Compared to the yielding strategy, speculation avoids the overhead incurred by context switches.

This breakdown indicates that the performance benefits from speculation hinges upon the time to create a speculative context and the amount of the system can safely speculate. If the cost is zero, and the device time (t_{device}) is short, then it is almost always better to speculate because the CPU can do useful work while the I/O is in progress, instead of spinning or paying the overhead of context switches. However, when t$_{device}$ is long compared to the time the system can safely speculate (t$_{spec\_execute}$), then yielding will perform better, since it can at least allow another application to do useful work where the speculation strategy would have to block. When the time to create a context (t$_{speculate}$) is high compared to t_{device}, then the blocking strategy would be better since it does not waste cycles creating a speculative context which will be committed before any work is done.

For millisecond devices, yielding is optimal because t_{device} is long, so the costs of scheduling and context switches are minimal compared to the time it takes to dispatch the I/O. For nanosecond devices, blocking is optimal since t_{device} is short, so overhead incurred by either speculation or yielding will be wasteful. For microsecond devices, we believe speculation could be optimal if

Application	Description
bzip2	bzip2 on the Linux kernel source.
dc	NPB Arithmetic data cube.
dd	The Unix dd utility.
git clone	Clone of the Linux git repository.
make	Build of the Linux 3.11.1 kernel.
mongoDB	A 50% read, 50% write workload.
OLTP	An OLTP benchmark using MySQL.
postmark	E-mail server simulation benchmark.
tar	Tarball of the Linux kernel.
TPCC-Uva	TPC-C running on postgresql.

Table 3: **Applications.** A wide array of applications which we analyzed for speculation potential.

there are microseconds of work to speculate across, and the cost of speculating is low.

3 The Potential for Speculation

In order for speculation to be worthwhile, $t_{spec\_execute}$ must be significantly large compared to the cost of speculation and the device time. In order to measure this potential, we instrumented applications with Pin [13], a dynamic binary instrumentation tool, to measure the number of instructions between I/O requests and the point speculation must block. For writes, we measured the number of instructions between a write system call and side effects-causing system calls (for example, `kill(2)` but not `getpid(2)`), as well as writes to shared memory. For reads, we measure the number of instructions between a read system call and the actual use of the resulting buffer (for example, as a result of a read instruction to the buffer), or other system call, as with a write. Our estimate of the opportunity for speculation is an extremely conservative one: we expect that we will have to block on a large number of system calls that many systems we discuss in section 4 speculate through. However, by limiting the scope of speculation, our findings reflect targets for speculation that produce a minimal amount of speculative state.

We instrumented a wide variety of applications (Table 3), and summarize the results in Table 2. In general, we found applications fell into one of three categories: pure I/O applications, I/O intensive applications, and compute intensive applications. We briefly discuss each class below:

Pure I/O applications such as `dd` and `postmark` performed very little work between side-effects. For example, `dd` performs a read on the input file to a buffer, followed by write to the output file repeatedly. On average, these applications perform on the order of 100 instructions between I/O requests and side effects.

We also looked at database applications including `TPCC-Uva`, `MongoDB` and `OLTP`. These applications are I/O intensive, but perform a significant amount of compute between side effects. On average, we found that these applications perform on the order of 10,000 instructions between read and write requests. These workloads provide an ample instruction load for microsecond devices to speculate through.

Compute intensive applications such as `bzip2` and `dc` performed hundreds of thousands to millions of instructions between side-effects. However, these applications made I/O calls less often than other application types, potentially minimizing the benefit to be had from speculation.

Many of the applications we tested used the buffer following a `read` system call immediately: most applications waited less than 100 instructions before using a buffer that was read. For many applications, this was due to buffering inside libc, and for many other applications, internal buffering (especially for the database workloads, which often employ their own buffer cache) may have been a factor.

4 Speculation Techniques

In the next section, we examine several techniques for speculation in the context of the microsecond era. We review past work and propose new design directions in the context of microsecond scale I/O.

4.1 Asynchronous I/O Interfaces

While asynchronous I/O interfaces [1] are not strictly a speculation technique, we mention asynchronous I/O since it provides similar speedups as speculation. Indeed, just as in speculation, program execution will continue without waiting for the I/O to complete. However, asynchronous I/O requires the programmer to explicitly use and reason about asynchrony, which increases program complexity. In practice, while modern Linux kernels support asynchronous I/O, applications use synchronous I/O unless they require high performance.

4.2 Software Checkpoint Speculation

In order to perform speculation, software checkpointing techniques generate a *checkpoint*, which is a copy of an application's state. To generate a checkpoint, we call `clone(2)`, which creates a copy-on-write clone of the calling process. After the checkpoint has been created, the system may allow an application to speculatively continue through a synchronous I/O call before it completes (by returning control to the application as if the I/O had completed successfully). The system then monitors the application and stops it if it performs an action which produces an external output (for example, writing

	Writes			Reads		
Application	Instructions	Calls/s	Stop Reason	Instructions	Calls/s	Stop Reason
Pure I/O Applications						
postmark	74 ± 107	518	close	15 ± 11	123	buffer
make (ld)	115 ± 6	55	lseek	8,790 ± 73,087	180	lseek (31%) buffer (68%)
dd	161 ± 552	697	write	69 ± 20	698	write
tar	248 ± 1,090	1,001	write (90%) close (9%)	144 ± 11	1,141	write
git clone	1,940 ± 11,033	2,833	write (73%) close (26%)	14 ± 10	1,820	buffer
I/O Intensive Applications						
MongoDB	10,112 ± 662,117	13,155	pwrite (94%)	62 ± 196	<1	buffer
TPCC-Uva	11,390 ± 256,018	115	write (49%) sendto (22%)	37 ± 8	22	buffer
OLTP	22,641 ± 342,110	141	pwrite (79%) sendto (7%)	31 ± 21	19	buffer
Compute Intensive Applications						
dc	1,216,281 ± 13,604,751	225	write	8,677 ± 66,273	156	buffer
make (cc1)	1,649,322 ± 819,258	12	write	165 ± 21	431	buffer
bzip2	43,492,452 ± 155,858,431	7	write	1,472 ± 345,827	18	buffer

Table 2: **Speculation Potential.** Potential for speculation in the read/write path of profiled applications. We list only the stop reasons that occur in >5% of all calls. Error numbers are standard deviations, and "buffer" indicates that speculation was stopped due to a read from the read buffer.

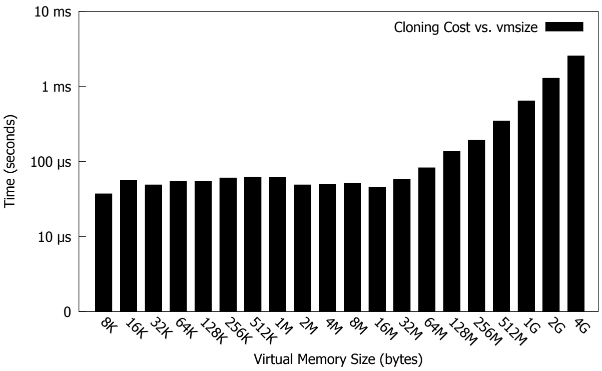


Figure 2: **Cloning Cost.** The cost of copy-on-write cloning for applications of various virtual memory sizes. Note that the axes are in log scale.

a message to a screen or sending a network packet) and waits for the speculated I/O to complete. If the I/O fails, the system uses the checkpoint created earlier to restore the application state, which allows the application to continue as if the (mis)speculation never occurred in the first place.

Software-based checkpointing techniques are at the heart of a number of systems which employ speculation, such as Speculator [9] and Xsyncfs [10]. These systems enabled speculative execution in both disks and over distributed file systems. These systems are particularly attractive because they offer increased performance without sacrificing correctness. Checkpoint-based speculation techniques hide misspeculations from the programmer, enabling applications to run on these systems unmodified.

However, providing the illusion of synchrony using checkpoints has a cost. We examined the cost of the clone operation, which is used for checkpointing (Figure 2). We found that for small applications, the cost was about 50μs, but this cost increased significantly as the virtual memory (vm) size of the application grew. As the application approached a vm size of 1GB, the cloning cost approached 1ms. While these cloning latencies may have been a small price to pay for slower storage technologies, such as disk and wide area networks, the cost of cloning even the smallest application can quickly eclipse the latency of a microsecond era device. In order for checkpoint-based speculation to be effective, the cost of taking a checkpoint must be minimized.

4.3 Hardware Checkpoint Speculation

Since we found checkpointing to be an attractive technique for enabling speculation given its correctness properties, creating checkpoints via hardware appeared to be a reasonable approach to accelerating checkpointing. Intel's transactional memory instructions, introduced with the Haswell microarchitecture [21] seemed to be a good match. Hardware transactional memory support has the potential of significantly reducing the cost of speculation, since speculative execution is similar to transactions. We can wrap speculative contexts into transactions which are committed only when the I/O succeeds. Checkpoints would then be automatically created and tracked by hardware, which buffers away modifications until they are ready to be committed.

We examined the performance of TSX and found that the cost of entering a transactional section is very low (<20 ns). Recent work [18, 21] suggests that TSX transaction working sets can write up to 16KB and <1 ms with low abort rates (<10%). While TSX shows much promise in enabling fast, hardware-assisted checkpointing, many operations including some I/O operations, cause a TSX transaction to abort. If an abort happens for any reason, all the work must be repeated again, significantly hampering performance. While hardware checkpoint speculation is promising, finer-grained software control is necessary. For example, allowing software to control which conditions cause an abort as well as what happens after an abort would enable speculation with TSX.

4.4 Checkpoint-Free Speculation

During our exploration of checkpoint-based speculation, we observed that the created checkpoints were rarely, if ever used. Checkpoints are only used to ensure correctness when a write fails. In a system with local I/O, a write failure is a rare event. Typically, such as in the case of a disk failure, there is little the application developer will do to recover from the failure other than reporting it to the user. Checkpoint-free speculation makes the observation that taking the performance overhead of checkpointing to protect against a rare event is inefficient. Instead of checkpointing, checkpoint-free speculation makes the assumption that every I/O will succeed, and that only external effects need to be prevented from appearing before the I/O completes. If a failure does occur, then the application is interrupted via a signal (instead of being rolled back) to do any final error handling before exiting.

Unfortunately, by deferring synchronous write I/Os to after a system call, the kernel must buffer the I/Os until they are written to disk. This increases memory pressure and requires an expensive memory copy for each I/O. We continue to believe that checkpoint-free speculation, if implemented together with kernel and user-space processes to allow omitting the memory copy, will result in a significant performance increase for microsecond-scale devices.

4.5 Prefetching

While the previous techniques are targeted towards speculating writes, prefetching is a technique for speculating across reads. In our characterization of speculation potential, we found that speculating across read calls would be ineffective because applications are likely to immediately use the results of that read. This result suggests that prefetching would be an excellent technique for microsecond devices since the latency of fetching data early is much lower with microsecond era devices, reducing the window of time that a prefetcher needs to account for. We note that the profitability of prefetching also decreases with latency – it is much more profitable to prefetch from a microsecond device that a nanosecond device.

Prefetching already exists in many storage systems. For example, the Linux buffer cache can prefetch data sequentially in a file. However, we believe that more aggressive forms of prefetching are worth revisiting for microsecond scale devices. For example, SpecHint and TIP [3, 11] used a combination of static and dynamic binary translation to speculatively generate I/O hints, which they extended into the operating system [4] to improve performance. Mowry [7] proposed a similar system which inserted I/O prefetching at compile time to hide I/O latency. Since microsecond devices expose orders of magnitude more operations per second than disk, these aggressive techniques will be much more lucrative in the microsecond era.

4.6 Parallelism

Other work on speculation focuses on using speculation to extract parallelism out of serial applications. For example, Wester [19] introduced a speculative system call API which exposes speculation to programmers, and Fast Track [6] implemented a runtime environment for speculation. This work will likely be very relevant since microsecond devices expose much more parallelism than disk.

5 Discussion

As we have seen, a variety of different techniques exist for speculating on storage I/O, however, in their current state, no technique yet completely fulfills the needs of microsecond scale I/O.

Our study suggests that future work is needed in two areas. First, more work is needed to design appropriate hardware for checkpointing solutions. Second, the opportunity for checkpoint-free speculation needs to be studied in depth for both compute intensive and I/O intensive database applications.

6 Conclusion

This paper argues for the use of speculation for microsecond-scale I/O. Microsecond-scale I/O will soon dominate datacenter workloads, and current strategies are suboptimal for dealing with the I/O latencies that future devices will deliver. Speculation can serve to bridge that gap, providing a strategy that enables I/O intensive applications to perform useful work while waiting for I/O to complete. Our results show that the performance of microsecond-scale I/Os can greatly benefit from speculation, but our analysis of speculation techniques shows that the cost of speculation must be minimized in order to derive any benefit.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. DGE-1144086, as well as C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[2] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 387–400. ACM, 2012.

[3] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 1–14, Berkeley, CA, USA, 1999. USENIX Association.

[4] K. Faser and F. Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *USENIX Annual Technical Conference, General Track*, pages 325–338, 2003.

[5] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, volume 91, pages 35–46, 1991.

[6] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A software system for speculative program optimization. In *Proceedings of the 7th annual IEEE/ACM International Symposium*, pages 157–168, 2009.

[7] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 3–17, New York, NY, USA, 1996. ACM.

[8] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th Conference on Parallel Architectures*, pages 135–145, 2004.

[9] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 24(4):361–392, 2006.

[10] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):6, 2008.

[11] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM, 1995.

[12] M. Qureshi, M. Franceschini, A. Jagmohan, and L. Lastras. Preset: Improving performance of phase change memories by exploiting asymmetry in write times. In *39th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391. IEEE, 2012.

[13] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education*, 2004.

[14] S.-H. Shin, D.-K. Shim, J.-Y. Jeong, O.-S. Kwon, S.-Y. Yoon, M.-H. Choi, T.-Y. Kim, H.-W. Park, H.-J. Yoon, Y.-S. Song, et al. A new 3-bit programming algorithm using SLC-to-TLC migration for 8MB/s high performance TLC NAND flash memory. In *VLSI Circuits (VLSIC), 2012 Symposium on*, pages 132–133. IEEE, 2012.

[15] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, 1981.

[16] H. Tanaka, M. Kido, K. Yahashi, M. Oomura, R. Katsumata, M. Kito, Y. Fukuzumi, M. Sato, Y. Nagata, Y. Matsuoka, et al. Bit cost scalable technology with punch and plug process for ultra high density flash memory. In *IEEE Symposium on VLSI Technology*, pages 14–15. IEEE, 2007.

[17] H. Volos. Revamping the system interface to storage-class memory, 2012. PhD Thesis. University of Wisconsin at Madison.

[18] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[19] B. Wester, P. M. Chen, and J. Flinn. Operating System Support for Application-specific Speculation. In *Proceedings of the Sixth European conference on Computer systems*, pages 229–242. ACM, 2011.

[20] 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*, 2012.

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel(R) Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 19. ACM, 2013.

OS I/O Path Optimizations for Flash Solid-state Drives

Woong Shin
Seoul National University

Qichen Chen
Seoul National University

Myoungwon Oh
Seoul National University

Hyeonsang Eom
Seoul National University

Heon Y. Yeom
Seoul National University

Abstract

In this paper, we present OS I/O path optimizations for NAND flash solid-state drives, aimed to minimize scheduling delays caused by additional contexts such as interrupt bottom halves and background queue runs. With our optimizations, these contexts are eliminated and merged into hardware interrupts or I/O participating threads without introducing side effects. This was achieved by pipelining fine grained host controller operations with the cooperation of I/O participating threads. To safely expose fine grained host controller operations to upper layers, we present a low level hardware abstraction layer interface. Evaluations with micro-benchmarks showed that our optimizations were capable of accommodating up to five, AHCI controller attached, SATA 3.0 SSD devices at 671k IOPS, while current Linux SCSI based I/O path was limited at 354k IOPS failing to accommodate more than three devices. Evaluation on an SSD backed key value system also showed IOPS improvement using our I/O optimizations.

1 Introduction

In recent years, solid-state drives (SSDs) have become more viable. Decrease in cost per GB and increase in performance made SSDs appealing to replace or reduce the usage of hard disk drives and even DRAM in the datacenter. Recent advances with the hardware interfaces such as SATA 3.0, SAS HD and PCI-Express 3.0 made these devices capable of even handling workloads which could only be served in main memory. However, the advance in SSD technology is not directly translated into user perceived performance. Software overhead is magnified as the performance of SSDs is enhanced [10, 7, 13, 14].

With the rise of faster memory technologies, such as DRAM or PCM, several studies were conducted on optimized I/O paths to mitigate these overheads [13, 6, 5, 14]. However, it is unclear how the enhancements would behave with the latencies of flash SSDs, which are orders of magnitude higher (vs DRAM or PCM) and highly unpredictable. Little work has been done with current generation of flash SSDs, especially in the I/O completion path.

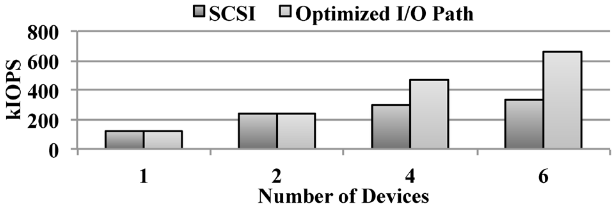


Figure 1: I/O performance of parallel 512-byte small random reads (Six SATA 3.0 SSDs attached to a desktop I/O chipset integrated AHCI controller)

In the I/O completion path, software delays caused by multiple context switches are considered harmful. These context switches are caused by additional I/O processing contexts such as interrupt bottom halves and background queue runs. In several studies for high performance storage, poll based synchronous I/O was used to eliminate these software delays [15, 6].

However, it is not trivial to apply this technique to NAND flash based SSDs. Flash SSD access latency is lower than hard disk drives, but it fails to go under ranges (under 20us) where poll can be beneficial. This even applies to high-end PCI-e flash devices [2].

In this paper, we present OS I/O path optimizations for flash SSDs, focused on minimizing software delays caused by the additional I/O processing contexts. With our optimizations, bottom half contexts and background queue running contexts are eliminated without introducing side effects. This is done by placing I/O operations in hardware interrupts or I/O participating parallel threads. Side effects of longer I/O processing delays are addressed by adopting a cooperative I/O processing model. All participating I/O threads actively share the burden of detecting I/O completions, performing I/O post processing and issuing new commands.

These I/O operations are exposed at a hardware abstraction layer which provides abstractions such as queues, tags and notifications commonly found in modern host controllers. The interface of the layer allows I/O threads to make synchronous decisions on whether to process pending I/O commands or not.

For evaluation, we implemented an I/O path based on our optimizations for an AHCI controller which can be considered as a worst case scenario for SSDs. We built a low cost system using six commodity SATA 3.0 SSDs connected to a single AHCI controller. With parallel 512-byte small random reads, our optimized I/O path was capable of accommodating up to five devices at 671k IOPS, while current Linux SCSI based I/O path was limited at 354k IOPS (Figure 1). Evaluation on an SSD backed key value system showed IOPS improvement using our I/O optimizations. Performance gain of our I/O path was 7% with the highest throughput (32 clients) and 108% under the highest load (256 clients).

2 Motivation

Whenever a new context (interrupt or thread) is introduced in the I/O path, scheduling delays, which can be significant on a busy CPU, are added to the I/O path (Figure 2). We were motivated to minimize these scheduling delays, which can be significant for SSDs, within the I/O path (Figure 3). However, it is not trivial to remove these contexts since these contexts are employed to maintain system responsiveness and system throughput. In the following, we state our motivations to remove these additional contexts and examine how they are employed in I/O paths for SSDs.

High IOPS, Smaller I/O: Software overheads, such as scheduling delays, can be minimized by issuing larger requests. However, bandwidth waste can be significant when the workload is oriented with high rates of small random requests. This motivated our work to remove these contexts. Parallel small random reads are gaining interest in the context of SSD backed key value storage which is considered as a good use case of SSDs [9, 1]. In this type of workload, a 30:1 `GET():SET()` ratio (read:write ratio) is observed, with 90% of values less than 500 bytes [3].

Conventional SCSI I/O Path: In many modern OSes, interrupt service handler routines (ISRs) are split into two parts to minimize system lockdown caused by heavy ISRs. This leads to at least two scheduling points during I/O completions. We show this in Figure 2. I/O thread 1 ((a) to (f) in Figure 2) shows the I/O completion path of Linux which is the I/O path for current SATA or SAS SSDs attached to AHCI controllers and SCSI based SAS controllers.

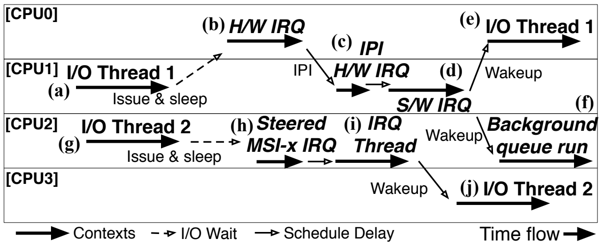


Figure 2: Scheduling delays in the I/O completion path

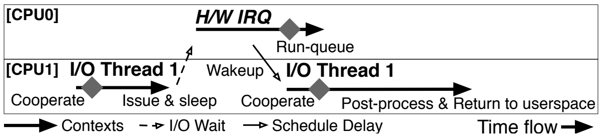


Figure 3: Minimizing scheduling delays within the I/O completion path

Software interrupts (d) are scheduled to relieve the main hardware interrupt handler (b) from I/O post processing tasks such as unmapping multiple DMA buffers and de-allocating I/O descriptors. To enhance CPU cache utilization of I/O post processing, software interrupts (d) are *steered* using inter processor interrupts (IPIs) [4] (c). In this case, an IPI to CPU core 1 is made to have I/O thread 1 (a) and the software interrupt (d) run on the same CPU. The background queue run context (f) is used to issue I/O requests which could not be issued immediately (i.e., a busy device).

Advanced Block Driver I/O Path: Recent NVM-Express standard [11] can simplify the I/O path with deeper (64k) queues and many (64k) queues. It is possible to eliminate queue runs and IPIs, but multiple scheduling delays within the completion path still exist.

In Linux, NVM-Express proposes a device driver [12] which bypasses the block layer (request queue) and the SCSI I/O subsystem. I/O Thread 2 ((g) to (j) in Figure 2) shows the I/O completion path of this driver. This driver performs direct issues to a deeper hardware queue, up to 64k in depth, which removes the necessity of the background queue run context (f). Also, it is possible to remove the use of IPIs for IRQ steering by having dedicated queue pairs (issue and completion) and interrupts (MSI/MSI-x) on CPU cores. IRQ handling is natively steered to designated CPU cores. Here, threaded interrupts are used, so software interrupts (d) are removed, but there are still delays of scheduling the IRQ thread (i) and scheduling the completion side of I/O thread 2 (j).

3 Design and Implementation

Our work was done to achieve the following goals: 1) Minimize scheduling delay by removing additional contexts, 2) Preserve the semantics of previous optimizations such as H/W IRQ relieving and IRQ steering, and 3) Generalize the optimizations to be applied to various host controllers.

To achieve these goals, we adopted a cooperative I/O processing model based on a set of fine grained operations exposed at a low level hardware abstraction layer (HAL). This HAL was introduced to generalize our optimizations to various host controllers.

3.1 HIOPS Hardware Abstraction Layer

The HAL, HIOPS-HAL (High IOPS - Hardware Abstraction Layer), has a role of exposing access and control of necessary H/W abstractions such as queues, tags and notifications implemented in the underlying H/W interface. These abstractions are commonly found in modern host controllers used for SSDs such as AHCI, NVM-Express, SCSI-Express and various SAS adaptors. Figure 4 shows the architectural role of the HAL which provides a generic interface to upper layers. The role is similar to the SCSI middle layer of Linux, though our HAL gives more access and control to upper layers.

Low Level Drivers: Similar to the VFS layer and the SCSI middle layer in Linux, the interface is implemented as a template of standard function pointers. Each entry of the template defines an operation, later invoked by an API call to the HAL. These API calls are implemented in Low Level Drivers (LLDs). Additionally, upper layer specific handlers are registered to LLDs for an upcall, and the upcall is done by LLDs at the point of notification. In this way, LLDs are capable of exposing execution contexts such as interrupts to upper layers. Details of the API calls and the upcalls are described in Section 3.2.

Interactions with Upper Layers: Beyond the HAL, an I/O strategy layer is responsible of mediating I/O requests from the upper layers to the HAL. In this work, we implemented an I/O strategy based on a cooperative I/O model. The I/O strategy is essentially a Linux block driver which provides a block interface to the rest of the system. While I/O strategies are not limited to expose block interfaces, the block interface was intended to limit upper layer modifications. Except for a few additional functions exposed for a modified VFS layer (Figure 4), all other block interface functions remain the same. At the top layer, ordinary `read()` and `write()` system calls are used to perform I/O, so applications can benefit from the I/O strategy optimizations without any modifications.

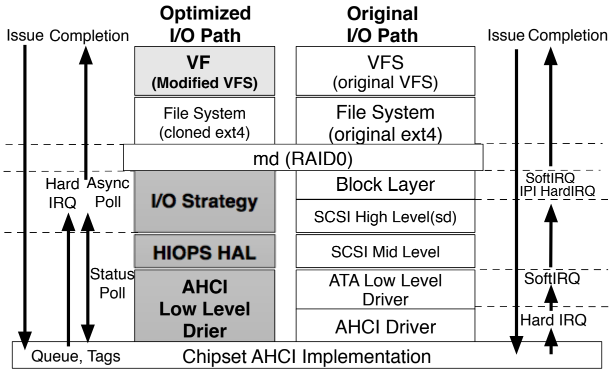


Figure 4: Comparison of our I/O path (left) and the original Linux SCSI I/O path (right)

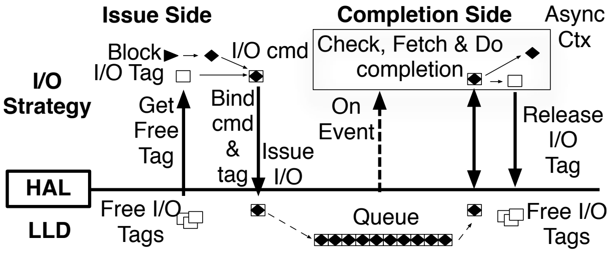


Figure 5: Interactions between I/O strategies and low level drivers (LLDs)

3.2 HAL API Operations

Figure 5 describes interactions between the upper layers and low level drivers (LLDs). The interactions consist of both issue side and completion side operations.

Issue Side Operations: In an I/O strategy, upper layer I/O requests are first converted to a low level command. Then a free tag (`get_free_tags`) is requested to be bound to the command (`bind_tag_cmd`). An implicit `begin_cmd` is called to timestamp the command (i.e., tracking I/O timeouts). Tags bound with commands are issued to the device by `issue_tags` calls. Note that tag related operations are named with plurals because they can be batched.

This interface gives flexibility to the I/O strategy so that it can synchronously determine whether the device is able to issue more I/O or not. If a `get_free_tags` call fails, then the device is busy.

Completion Side Operations: I/O strategies can decide whether to rely on interrupts. For interrupts, I/O strategies register a function pointer to gain synchronous access to the notification context. There, I/O strategies can check the I/O event status with `check_event` calls. Whenever there is an event, `fetch_event` is used to retrieve and process the command. If the I/O strategy does not rely on interrupts, it can synchronously check for

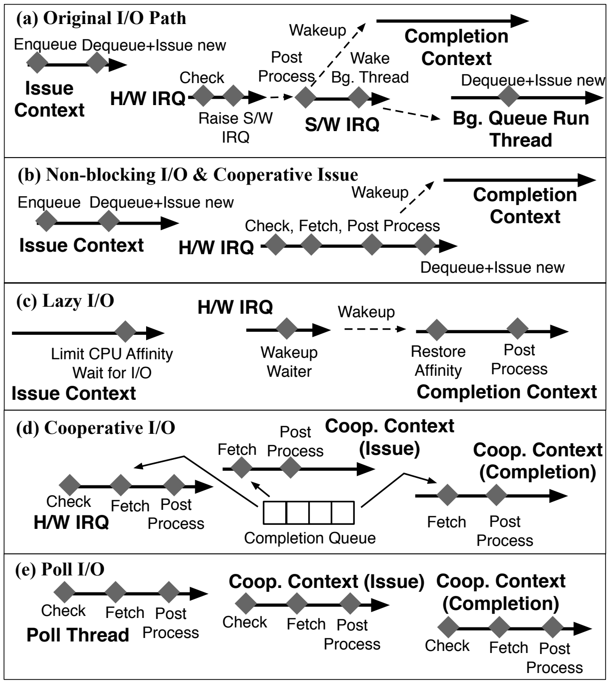


Figure 6: OS I/O path optimizations

events with the same process described above. In this case, the status check context is provided by the I/O strategy itself.

Completions are processed beginning with a detach_tag_cmd call to detach commands and tags. Detached tags are released to the controller with release_tags and the I/O strategy post processing contexts are initiated by end_cmd calls.

3.3 OS I/O Path Optimizations

In this Section, we describe OS I/O path optimizations based on a cooperative I/O model. These optimizations are implemented as a HIOPS-HAL I/O strategy which is mainly implemented as a Linux block driver.

Non-blocking I/O: No I/O contexts are blocked to acquire resources such as I/O tags without introducing additional background queue runs. In the issue path, all I/O commands are first enqueued into a simple software FIFO queue. If tags are available, a command is dequeued to be issued. Otherwise, the actual issue is deferred to other parallel issue paths or asynchronous contexts such as interrupts (Figure 6-(b)).

Here, the hardware interrupt context is used to issue remaining commands in the software FIFO queue. Since free I/O tags are *generated* in the hardware interrupt context, new I/O commands can be issued immediately using these free I/O tags without being blocked.

Lazy I/O Processing: Lazy I/O processing offloads I/O processing to the actual threads waiting for I/O completions (Figure 6-(c)). This eliminates additional context switches introduced by deferred I/O processing schemes such as bottom halves and threaded interrupts. This was done by exposing an alternative I/O-wait function from the I/O strategy to be called instead of the original io_schedule(). Here, a modified VFS layer calls this I/O-wait function to provide the contexts of I/O threads calling read() and write() system calls waiting for an I/O to complete. These I/O threads are blocked inside the provided I/O-wait function. Upon completion, these I/O threads are used for I/O post-processing instead of introducing additional contexts such as bottom halves and threaded contexts. I/O post processing is done by having HIOPS-HAL API calls after the I/O thread wakeup and before the I/O-wait function exits. After the I/O post processing, I/O threads return from the I/O-wait function and go back to the VFS layer and the userspace without any scheduling delays.

To enhance CPU cache hits during I/O post processing, waiters are awaken on CPUs where they issued the I/O and went to sleep. This is achieved by temporarily limiting the CPU affinity mask of a waiter thread to the current CPU before going to sleep. After the thread wakes up, the CPU affinity mask is restored.

Cooperative I/O Processing: Cooperative contexts are introduced by having HAL API calls from both the I/O issue path and the completion path (Figure 6-(d)) inside the I/O strategy. These are helper contexts which perform I/O tasks of other threads. All I/O threads voluntarily enter this cooperative context for every I/O request being frequently scheduled on the CPU. Here, I/O tasks can be carried out in a timely manner, even if the I/O owner thread is not being scheduled on the CPU. In a multi-core machine, the parallelism of I/O threads entering cooperative contexts increases overall I/O processing throughput of the system.

For cooperation, completion contexts make fetch_event calls to *steal* I/O processing work from post processing handlers. Issue threads performing non-blocking I/O issues for other I/O threads play another form of cooperation (Figure 6-(b)).

Poll Based I/O: Under higher loads, interrupts can be disabled. With interrupts disabled, a poll thread is introduced to poll for new I/O completions. Additionally, cooperative contexts are set to perform opportunistic poll (Figure 6-(e)). Note that polling is for the whole controller, not for individual I/O tags. Under high loads, the processing times of individual I/O commands are unpredictable, but the interval between multiple I/O commands completing in parallel is predictable (0us to 20us). After a single poll cycle, the poll thread releases the CPU and relies on high resolution timers to schedule the next poll. Poll thread introduces the overhead of timer inter-

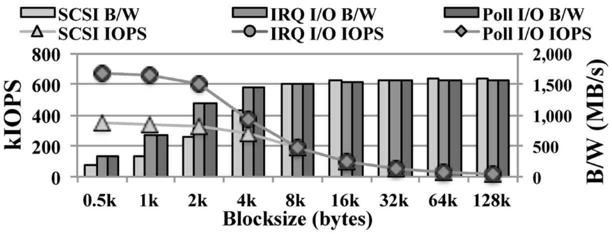


Figure 7: IOPS and bandwidth with increasing I/O block size (fio, Direct I/O, 128 threads, six SATA 3.0 SSDs, Software RAID0, 128kB stripe, ext4, noop scheduler(SCSI))

rupts, but the use of additional cooperative contexts lets us perform a rather coarse poll (16us to 32us).

It is possible to implement a hybrid mechanism to switch between the use of interrupts and poll methods, however the complication of determining mode switch leaves this implementation for our future work. Our experiments showed that indicators such as the current level of parallelism (occupied queue depth) combined with the current level IOPS can be a candidate to permit such tasks.

4 Evaluation

The impact of our optimizations was evaluated using a micro-benchmark application and a key value storage. fio 2.1.4 was used for our micro-benchmark evaluations, and Aerospike 2 was used as the key value system [1]. YCSB [8] was used to load the key value system.

Implementation: Our optimizations were applied to a Linux 3.2.40 kernel as dynamic loadable modules. These modules include the HAL itself, HAL I/O strategy, modified VFS and our custom AHCI HAL LLD (Figure 4). Here, the HAL consists of 6,187 lines of original code and the HAL I/O strategy with 1,766 lines of original code. The AHCI HAL low level driver was based on the AHCI SCSI libata device driver of Linux 3.2.40 but was modified to be a HIOPS-HAL LLD. Total 2,424 lines were original for HIOPS, and 2,632 lines were adopted from Linux 3.2.40. Modifications on the VFS layer was as small as 44 lines.

Experimental Setup: We conducted our evaluations on a PC with an Intel i7-4770 3.40Ghz hyper-threaded quad core CPU and 16GB DRAM. The system was equipped with six Samsung 840 Pro 256GB SATA 3.0 SSDs connected to a single AHCI controller which supports up to six SATA 3.0 ports.

I/O Throughput: Figure 7 shows the performance of our I/O paths with varying I/O blocksize. IRQ I/O was the hardware interrupt based I/O path presented in Section 3.3 and Poll I/O was the I/O path with interrupts disabled. With Poll I/O, the poll thread and the cooperative contexts performed poll altogether. All other optimizations were applied in both I/O paths described in Section 3.3.

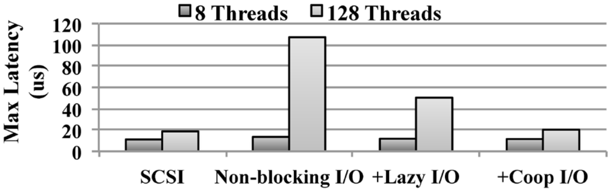


Figure 8: Effect of I/O processing optimizations

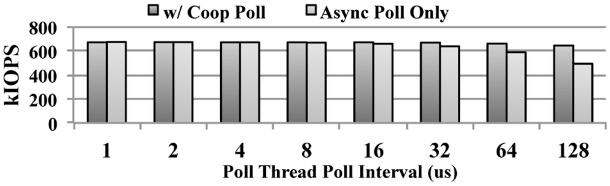


Figure 9: Poll interval impact

Our I/O paths achieve from 32% (4kB I/O) up to 89% (0.5kB I/O) IOPS gain over the original SCSI I/O path. IRQ I/O achieved 671k IOPS at maximum, while SCSI led 354k IOPS. However, there was no significant difference in IOPS between IRQ I/O and Poll I/O.

For requests larger than 8kB, there was no gain since the bandwidth was limited by the DMI 2.0 uplink bandwidth and the internal interconnects within the PCH. Our I/O paths with 4kB I/O were also limited by the the DMI 2.0 uplink bandwidth. The bandwidth converged to approximately 1.5GB/s which was similar to the bandwidth achievable from x4 PCI-Express 2.0 channels. This bandwidth was smaller than the DMI 2.0 25Gbps (2.5GB/s) uplink to the CPU.

I/O Post-processing Schemes: Figure 8 shows the effect of applying I/O post-processing schemes by showing the maximum latency of interrupt handlers. Under high loads (128 threads), basic Non-blocking I/O shows over 100us interrupt handling latency while the original SCSI I/O path shows up to 18us. This is because all I/O processing and next I/O issue had to be done in the hardware interrupt handler. When Lazy I/O is applied (+Lazy I/O), the latency diminishes to 50us. With both Lazy I/O and Coop I/O applied (+Coop I/O), the maximum latency drops to 20us which is similar to the original SCSI I/O path.

Polling: Figure 9 shows the impact of the poll interval. The load was 128 threads performing 512-byte direct I/O (O_DIRECT) read()s. 'Async poll' was with a single poll thread polling, and 'w/ Coop Poll' was with opportunistic completion checks in both issue and completion paths helping the poll thread. The results show that coopera-

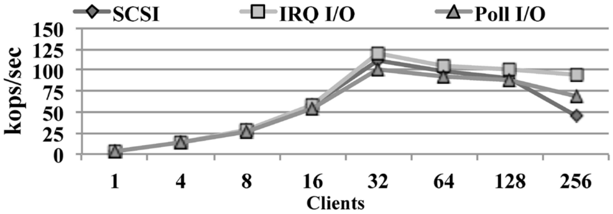


Figure 10: Key value storage performance (YCSB 100% `get()` performance)

tive poll was capable of over 600k IOPS even if the poll interval increased up to 128us.

Key Value Storage: We evaluated our I/O paths under an SSD backed key value storage. For this evaluation, another identical Intel i7 quad core CPU system was linked back to back through a pair of 1Gbps NICs. To minimize the storage latency, this key value storage did not use file systems when it performed storage I/O. Also, all I/O was performed with direct I/O (`O_DIRECT`) to eliminate page cache incurred overheads. In our evaluation, six SSDs were used by the key value storage.

Figure 10 shows key value throughput with increasing YCSB client threads. While performance with SCSI I/O degrades beyond 128 clients, our optimizations were able to mitigate the collapse. The highest throughput was 119kops/sec achieved with IRQ I/O while SCSI I/O showed 110kops/sec and Poll I/O showed 101kops/sec. Poll I/O showed lower performance than SCSI I/O, because the storage throughput was not high enough relative to the storage throughput seen in Figure 7. This motivates a poll & IRQ hybrid I/O scheme. The key value storage could only load the storage up to 150k IOPS at its peak. Performance gain of IRQ I/O over SCSI was 7% with the highest throughput (32 clients) and 108% under the highest load (256 clients).

5 Conclusion

Previous I/O completion schemes for fast storage are not sufficient to support current flash SSDs. With faster memory technologies, the software delays of multiple context switches can be mitigated with techniques such as polling; however, the noncommittal latencies of flash SSDs, not like DRAM nor like hard disks, require the use of different approaches in addition to such a technique.

In this paper, we have presented a low latency I/O completion scheme based on a cooperative I/O processing model. Additional I/O contexts such as bottom halves and background queue runs are eliminated, and their absence is compensated with the opportunistic help of I/O participating threads under parallel high IOPS workloads. I/O workloads with low parallelism can enjoy the lower latency of the our simplified hardware interrupt based I/O post processing. Our evaluation on an SSD backed key value storage suggests that workloads of high I/O parallelism will benefit from our I/O completion scheme.

Acknowledgments

We would like to thank the anonymous USENIX ATC reviewers and our shepherd Kai Shen for comments that helped improve this paper. This work was supported by the National Research Foundation of Korea (NRF) grants 2010-0020731 and NRF-2013R1A1A2064629. The ICT at Seoul National University provided research facilities for this study.

References

[1] Aerospike. aerospike2 free community version. `http://www.aerospike.com`.

[2] Fusion-io. ioDrive II. `http://www.fusionio.com/products/iodrive2/`.

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. SIGMETRICS'12, ACM.

[4] AXBOE, J. Io queuing and complete affinity. `http://lwn.net/Articles/268713/`, Feb 2008.

[5] BJØ RLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block IO: introducing multi-queue SSD access on multi-core systems. SYSTOR'13, ACM.

[6] CAULFIELD, A., MOLLOV, T., AND EISNER, L. Providing Safe, User Space Access to Fast, Solid State Disks. ASPLOS'12, ACM.

[7] CAULFIELD, A. M., COBURN, J., MOLLOV, T., DE, A., AKEL, A., HE, J., JAGATHEESAN, A., GUPTA, R. K., SNAVELY, A., AND SWANSON, S. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. SC'10, IEEE.

[8] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. SoCC'10, ACM.

[9] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: high throughput persistent key-value store. VLDB'10, VLDB Endowment.

[10] FOONG, A. P., VEAL, B., AND HADY, F. T. Towards ssd-ready enterprise platforms. ADMS'10.

[11] HUFFMAN, A. NVM Express specification 1.1a. `http://www.nvmexpress.org/specifications/`, September 2013.

[12] NVM-EXPRESS. `http://www.nvmexpress.org/resources/linux-driver-information/`.

[13] SEPPANE, E., O'KEEFE, M. T., AND LILJA, D. J. High performance solid state storage under Linux. MSST'10, IEEE.

[14] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. SoCC'12, ACM.

[15] YANG, J., MINTURN, D. B., AND HADY, F. When Poll is Better than Interrupt. FAST'12, USENIX.

FlexECC: Partially Relaxing ECC of MLC SSD for Better Cache Performance

Ping Huang[‡§], Pradeep Subedi[‡], Xubin He[‡✉], Shuang He[§], Ke Zhou[§]
[‡]*Virginia Commonwealth University, USA*
[§] *Wuhan National Laboratory for Optoelectronics*
Huazhong University of Science and Technology, China
{*phuang,subedip,xhe2*}*@vcu.edu, hshopeful@gmail.com,k.zhou@hust.edu.cn*

Abstract

The ever-growing capacity and continuously-dropping price have enabled flash-based MLC SSDs to be widely deployed as large non-volatile cache for storage systems. As MLC SSDs become increasingly denser and larger-capacity, more complex and complicated Error Correction Code (ECC) schemes are required to fight against the decreasing raw reliability associated with shrinking cells. However, sophisticated ECCs could impose excessive overhead on page decoding latency and thus hurt performance. In fact, we could avoid employing expensive ECC schemes inside SSDs which are utilized at the cache layer. We propose *FlexECC*, a specifically designed MLC SSD architecture for the purpose of better cache performance without compromising system reliability and consistency. With the help of an upper-layer cache manager classifying and passing down block access hints, *FlexECC* chooses to apply either regular ECC or lightweight Error Detection Code (EDC) for blocks. To reduce performance penalty caused by retrieving backend copies for corrupted blocks from the next-level store, *FlexECC* periodically schedules a scrubbing process to verify the integrity of blocks protected by EDC and replenish corrupted ones into the cache in advance. Experimental results of a proof-of-concept *FlexECC* implementation show that compared to SSDs armed with regular ECC schemes, *FlexECC* improves cache performance by up to 30.8% for representative workloads and 63.5% for read-intensive workloads due to reduced read latency and garbage collection overhead. In addition, *FlexECC* also retains its performance advantages even under various faulty conditions without sacrificing system resiliency.

1 Introduction

As system architects have always been pursuing to build and/or optimize storage systems in both high-performance and cost-effective ways, NAND flash-based Solid State Drives (SSD) have been intensively researched to be efficiently utilized in various storage systems during the past decade due to their highly desirable characteristics (e.g., high performance and low power consumption) [5, 35, 34, 39]. Compared to rotating hard disk drives (HDD), SSDs provide one order of magnitude higher performance while consuming much less power to finish running the same workloads[13, 20].

However, because of the relatively high cost per GB [30, 12] and limited lifetime concerns [2, 26, 19], NAND flash-based SSDs are nowadays particularly widely utilized as a cache in front of storage systems comprised of HDDs, aiming to exploit their complementary advantages [36, 4, 18]. For instance, SSDs have already been utilized as front-end caches in storage products, including EMC's *VFCache* [1], Apple's *Fusion Drive* and Fusion's *ioControl*^{TM} *Hybrid Storage* and deployed in various scenarios including networked environment [22], cloud infrastructures [27, 4].

The enabling factors of SSD's wide deployment as a large non-volatile cache are primarily attributed to their steadily-expanding capacity and the resultant affordable cost, which in turn are essentially driven by technology scaling and the employment of Multi Level Cell (MLC), i.e, scientists have pushed two or even more bits into each diminishingly-sized flash memory cell. Researchers have suggested a variety of ways to improve SSD cache performance according to flash peculiarities, including dividing the cache space into read and write caches [21] and designing effective cache algorithms [42]. However, technology scaling has also caused many concerns [15]. For example, recent research findings have revealed that increasing an additional bit in a storing cell would reduce chip's lifetime by 5-10%, and shrink throughput and increase latency by 55% and 2.3x on average, respectively [16]. These problems could become an impediment to further improving their performance as a cache which is supposed to provide high performance.

Each NAND flash memory cell is a floating gate transistor that is able to preserve electrons. Bits information stored in each flash memory cell are represented and differentiated by the different voltage levels of the trapped charges. The reliability-impacting factors such as programming inaccuracy, electron detrapping, cell-to-cell interference [33, 32] are becoming increasingly severe when cells are pushed to store more bits, causing flash memory to exhibit an climbing high raw bit error rate (RBER) [15, 16], which renders them not suitable for practical usage. For example, the RBER of MLC flash memory is around 10^{-6}, while manufacturers usually rate the uncorrectable bit error rate (UBER) in their data sheets to be 10^{-11} [29, 10]. To bridge the reliability gap, sector-level or page-level Error Correcting Codes are synergistically implemented in flash memory controllers to achieve a practically acceptable UBER. However, as flash storage gets denser, we have witnessed the deployed ECCs are becoming more and more advanced and complicated [7], from Hamming Code to Bose-Chaudhuri-Hocquenghem (BCH) and Reed-Solomon codes [10] to Low Density Parity-check (LDPC) [49]. Therefore, the ECC implementation complexity has correspondingly increased significantly, causing prolonged encoding and decoding latencies. For instance, the decoding latency of a BCH tolerating 12 bit errors for an 8KB page are around $180\mu s$ and $17\mu s$ in contrast to $90\mu s$ and $10\mu s$ in a BCH tolerating 6 bit errors, with software [21] and hardware [41] implementation respectively, which accounts for a significant percentage of the page access latency.

Fortunately, in cache-oriented MLC flash-based SSDs, the need for expensive error correcting codes could be obviated. The reason is that in occurrences of errors, accesses to corrupted blocks can be serviced by their backup copies in the next layer and most of the time accesses can be completed faster because of the absence of excessive decoding overhead. Based on this observation, in this paper, we advocate a novel cache-oriented SSD architecture called *FlexECC*, a cross-layer design targeting specifically for MLC SSDs exhibiting high RBER and employing expensive ECC schemes. *FlexECC* achieves better cache performance by flexibly and selectively applying either regular ECC or light EDC [21] to flash pages according to their consistency and reliability requirements. Specifically, taking advantage of the information conveyed by the frontend cache manager via proposed interfaces (Section 3.3), *FlexECC* can easily identify the storage requirements of different pages and accordingly apply the appropriate protection or correction schemes via programmable flash memory controller. When writing fresh data which have no backup copies in the next storage layer, *FlexECC* adopts normal error correction code (specifically, BCH in our design), otherwise it applies simple and fast error detection code (EDC) (specifically, cyclic redundancy code or CRC in our design). Due to the differences in decoding latencies between BCH and CRC (Section 2.2), read accesses to CRC-protected pages would be significantly speeded up, enhancing the cache performance. The more CRC-protected pages in the cache device, the greater cache performance *FlexECC* provides. Furthermore, in order to mitigate the performance impacts of fetching data from the underlying layer for corrupted pages in the critical path, *FlexECC* schedules a scrubbing process to verify the integrity of CRC-protected pages and populates corrupted pages in advance. Evaluation results with both representative and synthetic workloads have shown that *FlexECC* is able to improve cache performance by an impressive degree without sacrificing consistency and reliability relative to normal ECC-armed MLC flash-based SSD.

Our main contributions in this work are two-fold. First, to the best of our knowledge, the proposed *FlexECC* is the first work to selectively replace ECC with EDC to improve SSD cache performance without compromising cache consistency and reliability. This bears important implications for future-generation MLC SSDs which require more advanced error correction codes and incur high decoding latencies to read operations. We are not making trade-offs between performance and reliability or consistency [25, 32], instead, we aim to improve performance while maintaining the same level of resilience by leveraging the characteristics of cache systems. Second, we have implemented a proof-of-concept prototype of *FlexECC* and conducted extensive evaluations to show that *FlexECC* is able to improve performance over conventional SSDs for a variety of workloads, even under various faulty conditions.

The remainder of this paper proceeds as follows. We discuss the background and our motivation in Section 2. Following that, we elaborate on the details of *FlexECC* in Section 3. We conduct experiments to evaluate *FlexECC* in Section 4, followed by a discussion of related work in Section 5. Conclusions of this work are given in Section 6.

2 Background and Motivation

2.1 Flash Memory Reliability

Flash memory cells are floating gate transistors which hold electrons to represent information. Each flash memory cell can be designated to represent one bit information (SLC), two bits (MLC) or three bit (TLC). The represented storage state is differentiated by the voltage level of trapped charges. Programming or writing flash memory cell is the process of injecting electrons into the cell to the level corresponding to the desired state, and reading is the process of sensing out the represented voltage level and comparing it with preset reference levels to determine its value. By its very nature, the trapped charges are constantly in a moving state and can shift to their neighboring cells (i.e., current leakage) over time, causing voltage shifting [32, 25] and therefore data corruption. Moreover, as flash memory cells experience more program and erase operations, their charge-trapping ability degrades and as a result are more prone to errors [33]. The occurring probability of these errors are called raw bit error rate (RBER). SLC flash memory typically exhibits two orders of magnitude better RBER than MLC [14, 10], because MLC flash memory has much shorter differential voltage window between adjacent voltage thresholds than SLC, which causes it more difficult for MLC to differentiate the statuses.

In design practice, flash-based SSDs typically implement ECCs in memory controllers [21] to meet reliability and endurance requirements, causing a performance-reliability trade-off in the design space. ECC is a kind of information encoding scheme which can tolerate a specified number of bit errors (called error correction capability *t*) by augmenting a certain amount of redundant information to the original message of length *k*, which typically equals to the page size in flash memory. Corrupted message can be reconstructed via decoding as long as the number of bit corruptions are within the ECC correction capability. Considering the wide adoption of BCH code in commercial SSDs, we base our discussions on BCH in the remaining sections. The bit error rate after applying ECC is called uncorrectable bit error rate (UBER). Assume an ECC scheme has an error corruption capability of *t* and the length of an encoded message is *N*, then the relationship between UBER (P_{UBER}) and RBER (P_{RBER}) is given by Equation 1.

$$P_{UBER} = \frac{\sum_{n=t+1}^{N} \binom{N}{n} * (P_{RBER})^n * (1 - P_{RBER})^{N-n}}{N} \quad (1)$$

Intuitively, to guarantee the same level of UBER (e.g., 10^{-11}), we can either increase ECC correction capability or decrease RBER. For example, more precise Incremental Step Pulse Programming control (i.e., using smaller $\triangle V_{pp}$ [44, 43]) produces smaller RBER and using more powerful ECC schemes also guarantees target reliability [6, 9]. However, as flash geometries become increasingly smaller (3x- and 2x-nm regimes) and denser [11], those techniques are no longer necessarily as effective as before, which is evidenced by the continuous increases in error correction requirements, program time and read time observed between different flash process generations [7]. When flash storage becomes denser, the noise margin narrows, necessitating very small $\triangle V_{pp}$ to program pages and thus causing prolonged programming process and imposing significant overhead on performance [33, 32]. On the other end, implementing more powerful ECC schemes on denser flash memory could be prohibitively expensive or even unrealistic for the following reasons. First, correction logic becomes complex, costly and occupies more silicon area and the resultant decoding latency increases correspondingly. Second, it increases power dissipation, whose side effects counteract ECC's efforts to improve reliability. Third, the page spare area may no longer have enough space for the expanding redundant information.

2.2 Replacing ECC with EDC for Cache

It has been observed in previous research [33, 25] that most of the occurred errors in flash memory are retention errors, i.e., errors caused by loss of charges over time, and flash memory exhibits reasonably high reliability at its early usage. In contrast to permanent storage, cached data are transient and live for a short lifespan, typically ranging from seconds or hours to days rather than months or years [48]. Therefore, the corruption probability of cached data is comparatively low. Moreover, even if corruptions do occur to cached data, corrupted data blocks can still be serviced by back-end storage as long as the blocks have been flushed down beforehand, at the cost of accessing disk or RAID storage systems.

Based on the above analysis, we are motivated to selectively relax ECC correction capability of certain cache blocks (i.e., the blocks which have consistent backup copies) to avoid decoding overhead for better performance. We only reserve error detection capability using lightweight EDC for ECC-relaxed blocks. Given the low corruption probability of short-living cached data and the significant discrepancy between ECC decoding latency and EDC verification overhead (to be discussed shortly), it is reasonable to expect performance improvement coming out of ECC-relaxed cache architectures while maintaining system reliability.

In the remainder of this section, we conduct theoretical performance analysis on ECC and EDC to demonstrate the potential performance gains that could be obtained by replacing ECC with EDC. Due to their popularity, we use primitive binary BCH [40, 41] for default ECC and CRC for default EDC. CRC, short for cyclic redundancy code, is an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data. The detection capability of CRC is characterized by how many concurrent bit errors it is able to detect. Binary BCH code has a form of (n,k,t), where n is the codeword length equal to $2^m - 1$ for some positive integer *m*, *t* is the correction capability indicating the maximum bit errors BCH is able to tolerate, and *k* is the length of the original message. The BCH arithmetic operations are based on Galois field GF(2^m) [24]. For fairness, we configure BCH to be able to tolerate t bit errors and CRC to detect t bit errors for each flash page[1].

Encoding: Encoding operation is associated with every write operation in flash to calculate redundant bits which are then written to the flash page's spare area together with page data. CRC and BCH share the same simple encoding procedure. To encode a page message M(x), both CRC and BCH divide the original message by a polynomial generator G(x) whose degree is dependent on t. The resultant remainder R(x) is the redundant information. Equation 2 gives the encoding calculation. Therefore, the encoding latencies of both CRC and BCH are approximately the same and equal to the time taken by a polynomial multiplication [8]. In other words, CRC and BCH incur the same additional latency to write operations.

$$\frac{M(x)}{G(x)} = Q(x) + \frac{R(x)}{G(x)} \tag{2}$$

Decoding/Detecting: A decoding/detecting process accompanies every flash page read operation. After reading out each page content, the flash memory controller verifies the integrity of the page content according to the adopted protection scheme. In contrast to the similar encoding procedure, BCH decoding is far different from CRC detection. CRC detection process is quite straightforward. Suppose the read page content is M(x)$'$. CRC performs the same arithmetic operation as in Equation 2 and checks whether the new remainder is identical to the previous one or not. Essentially, it is equivalent to verifying Equation 3. If Equation 3 holds true, then it is assumed no error occurs, otherwise the message is considered corrupted, so CRC detection process consumes the same time as CRC encoding.

$$\frac{M(x)' - R(x)}{G(x)} - Q(x) = 0 \tag{3}$$

BCH decoding is much more complicated and involves three steps, *syndrome computations*, *finding error-location polynomial* and *error correction*. The first step is to compute $2t$ syndrome components $S_1, S_2, \cdots, S_{2t}$, each of which is essentially a polynomial calculation. Then, based on the $2t$ syndromes, the second step uses *Berlekamp-Massey* algorithm to calculate the error-location polynomial $\sigma(x) = 1 + \sigma_1(x) + \sigma_2(x^2) + \cdots + \sigma_t(x^t)$. Finally, the third step solves the roots of $\sigma(x) = 0$ by using exhaustive *Chien Search* algorithm and outputs an error vector indicating the error positions. It should be noted that after obtaining the $2t$ syndromes, if all of them are evaluated to zeros, the message is considered error-free and the decoding process terminates immediately. Specific details about BCH code can be found in [24].

According to [24], a polynomial calculation takes $(n-1)$ additions and $(n-1)$ multiplications, *syndrome computations* take $(n-1)t$ additions and nt multiplications, *finding error-location polynomial* takes $2t^2$ additions and $2t^2$ multiplications, and *error correction* takes nt additions and nt multiplication. Suppose T_a and T_m are the time needed by per addition and per multiplication, respectively. Then the achievable speedups of replacing BCH with CRC for decoding a correct message ($S_{correct}$) and a corrupted message (S_{error}) are given by Equation 4 and Equation 5, respectively.

$$S_{correct} = \frac{(n-1) \times t \times T_a + n \times t \times T_m}{(n-1) \times (T_a + T_m)} \tag{4}$$

$$S_{error} = \frac{(2t^2 + 2nt - t) \times T_a + (2t^2 + 2nt) \times T_m}{(n-1) \times (T_a + T_m)} \tag{5}$$

Suppose $T_m = T_a$, i.e., the time taken to perform an addition is equal to that of a multiplication[2], typically one clock cycle, then $S_{correct}$ and S_{error} become $\frac{(2n-1)t}{2(n-1)}$ and $\frac{4nt+4t^2-t}{2(n-1)}$, which in turn approximately approach t and $2t$, respectively, when $n \gg t$.

In summary, we have demonstrated that by using CRC instead of BCH, we are able to reduce the latencies of decoding uncorrupted and corrupted message by t and $2t$ times, respectively. Given the increasing value of t and the associated decoding latency in MLC SSDs, the extent of decoding latency reduction will increase correspondingly and potentially translate to more significant performance improvement.

In real implementations, BCH can be either realized in software or hardware. In this paper, we assume hardware implementation, since typically the memory controller inside SSDs employs electrical circuit to perform BCH encoding and decoding for high performance purpose. BCH hardware implementation presents trade-offs among chip area, cost and latency [41, 40]. Different implementation configurations would cause different latencies. The more circuits are deployed, the less latency it incurs, but the more energy it consumes. In our evaluations, according to the hardware implementation in [41], we use $10\mu s$ as the decoding latency of a BCH tolerating 6 bit errors out of a flash page. Based on this latency, we use the above analysis to derive other parameters as shown in Table 2 in Section 4.1.

3 *FlexECC* Design and Implementation

In this section, we elaborate on the design and implementation details of *FlexECC*. We first give an overview of *FlexECC*, followed by a basic description of the cache manager in which we collect and pass down the block access information. Then we present the proposed extended interfaces via which access information is passed down to facilitate the underlying device's internal management. Following that, we describe the scrubbing process which is a precautional technique to suppress the performance overhead associated with accesses to erroneous pages. Moving on, we briefly discuss the garbage collection process in *FlexECC* with a focus on the differences relative to conventional SSDs. Finally, we give a holistic discussion on how the incoming requests are handled by *FlexECC*.

3.1 System Overview

As discussed previously, the idea of *FlexECC* is quite simple. It essentially comes down to two critical problems. The first problem is how to characterize block access behaviors and relay the collected information to the underlying device. The second problem is how the cache device can take advantage of the collected information to improve its performance. For the first problem, *FlexECC* augments a cache monitor into an ordinary cache manager. The monitor observes the cache behaviors and infers the storage requirements of the corresponding blocks which are are supposed to be stored in the cache layer. For the second problem, *FlexECC* employs a *Programmable Memory Controller (PMC)* inside the cache device to dynamically allocate CRC-protected or BCH-encoded pages to accommodate the incoming page writes according to their storage requirements.

Figure 1 shows a holistically architectural view of *FlexECC*. As depicted in the picture, the upper part is a modified cache manager which is able to collect block access information and send the information down to the SSD cache device to facilitate its internal management. In the middle is the SSD cache device with two added components including a hardware PMC and a software scrubber. In the bottom is the underlying storage system comprised of HDDs. In addition to constructing a basic hybrid storage system, the cache manager is augmented with the functionality of tracking and tagging block accesses to SSD. The collected information can be passed down to SSD via extending cross-layer interfaces, which has been proposed and evidenced by the techniques employed in previous researches including *Shepherding I/O* [17], *DSS* [28] and *FlashTier* [37]. The SSD cache device internally employs a *Programmable Memory Controller (PMC)* [21] which is able to program pages[3]to be either BCH-encoded or CRC-protected and allocate different types of pages to accommodate incoming requests according to their respective requirements, which is in spirit similar to the fast and slow pages allocation policy described in [16]. The *PMC* divides the entire cache space into two different regions, namely, CRC-region and BCH-region. Moreover, *FlexECC* actively initiates a scrubber process to verify the integrity of CRC-protected pages and prepares to populate corrupted ones from underlying storage before they are accessed. In addition, the SSD FTL is slightly modified, with each FTL entry having several added tags to provide auxiliary information, for example, in what code scheme (BCH or CRC) the page is protected, etc.

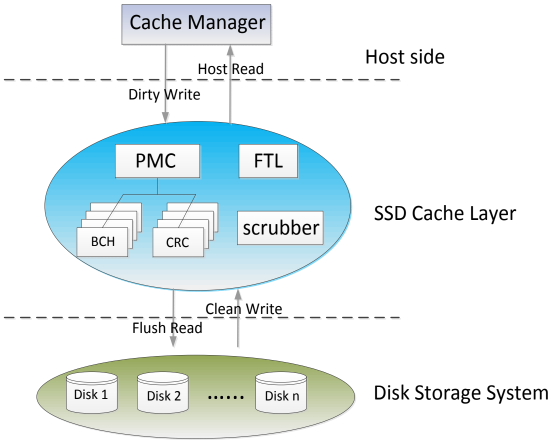


Figure 1: System Architecture.

3.2 Cache Manager

In our context, a *cache manager* interposes above the disk device driver in the operating system to send requests to either the flash device or the disk system directly, as it is with *FlashTier* [37]. It transparently constructs and manages an SSD-HDD hybrid storage system in which SSD acts as an inclusive cache layer above the disk system. The *cache manager* dictates which data blocks are written to the SSD cache through the adopted cache replacement policy, like Least-Recently-Used (LRU) or First-In-First-Out (FIFO). It supports two cache modes: *Write-through* and *Write-back*. In *Write-through* mode, on every write to the SSD cache, the cache manager also persists the data to the disk system before it reports write completion, which guarantees consistence at any time. In *Write-back* mode, the cache manager may write to the SSD cache without updating the disk system, causing dirty data in the cache. Cached blocks are flushed to the disk system for persistence at a configurable rate, e.g., every 5 minutes. For a write request, when there is no enough space, the cache manager evicts a victim block according to the replacement policy to make room for the incoming write. For a read request, the cache manager first consults the SSD. If it is not present in the cache, the cache manager fetches the data from the underlying disk system and populates it into the cache. By default, we assume *Write-back* mode in the discussion of *FlexECC*, because *Write-through* is an extreme scenario in which the whole cache space could be safely CRC-protected. We implement the *cache manager* based on *FlashCache* [37]. Specifically, we monitor every triggering event that causes read or write operation to the SSD cache, and forward the information to SSD via extended access interfaces.

3.3 Extended Interfaces

Extending existing interfaces between neighboring layers to communicate useful information for various purposes has been proposed in previous literature [28, 38]. Such extensions can be conveniently realized via leveraging the reserved or unused bits in the communication protocols, e.g., SCSI protocol. In *FlexECC*, we use a similar approach to pass information about cache behaviors to SSD to help its internal management. We propose four extended access interfaces to capture different reasons that cause accesses to the SSD, namely *Dirty Write*, *Host Read*, *Clean Write* and *Flush Read*, which are indicated in Figure 1. These interfaces are defined from the perspective of the SSD cache device. We discuss each of them as follows:

Dirty Write: a request to write a fresh data block which has no backup copy in the disk system and thus requires high reliability guarantee. New content generated by upper-level applications are written into the cache device using this interface. In response to this operation, the programmable memory controller designates BCH-encoded pages to store the content.

Clean Write: a write request to write a clean data block which has consistent backup copy in the back-end storage. Block migrations originating from disk to cache, e.g., due to a miss or populating corrupted pages, are accomplished through this interface. In response to this operation, the programmable memory controller designates CRC-protected pages to store the content.

Host Read: This interface is used to satisfy data reads issued by applications. It corresponds to cache read hit. *Host Read* data can be either BCH-encoded or CRC-encoded, depending on its state when it is requested.

Flush Read: a read caused by flushing dirty data back to the disk system due to releasing cache space or periodical time-out flushing down. Internally, *FlexECC* monitors this operation and marks associated flash pages as eligible to be free from BCH-encoded. During garbage collection, the marked pages contained in victim blocks are relocated to clean blocks using *Clean Write* interface.

In *FlexECC*, a data block could be in four states, which are named *HOST*, *BCH*, *CRC*, and *HDD*, indicating when the block is in host memory, in a BCH-encoded page, in a CRC-protected page and in back-end storage, respectively. However, it should be noted that these four states are not necessarily exclusive to one another. For example, the same block can be in *HDD* and *CRC* states simultaneously. The data block changes its state in response to the operations applied to it. Figure 2 shows the page state transition diagram with respect to the interface operations. The *GC write* represented by dashed arrow denotes the state transition of marked data pages (by *Flush Read*) from *BCH* to *CRC* during garbage collection.

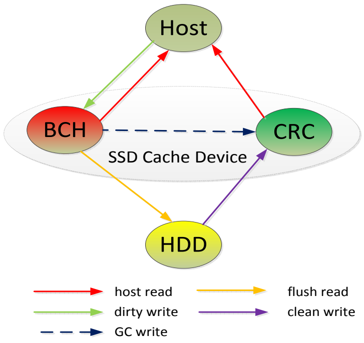


Figure 2: Page Content State Transition Diagram. The page content can be in the host memory (HOST), protected by BCH code (BCH), protected by CRC code (CRC) or in the back-end storage (HDD). Read and write are defined in respect to the cache device.

3.4 Disk Scrubber

While CRC can guarantee the minimum reliability, due to CRC's inability to correct errors, corruptions occurred to CRC-encoded pages may incur extraordinarily large overhead, since accesses to corrupted pages have to be satisfied by the back-end storage whose performance is typically one order of magnitude slower than that of the cache. To prevent such overhead severely impacting performance, *FlexECC* regularly schedules a *Scrubber* [31] process to verify the integrity of CRC-encoded pages. The *Scrubber* is a two-step process. The first step is a lightweight step which iteratively scans the CRC-protected pages to verify their checksums. If any inconsistency is detected, the corresponding page is marked as corrupted. The second step is to initiate a data migration process to replenish those corrupted pages found in the first step. Depending on when the corrupted pages are accessed, they could be forced to be fetched from back-end storage on the access path to them, which could cause significant performance overhead, or they could be prefetched by *Scrubber* into the cache before they are actually accessed. To minimize performance impacts, we leverage the idleness in the workloads to launch the scrubber process. Specifically, when the observed inter-request interval T_{inter} is longer than a configured multiple m of the time T_{crc} taken by CRC verification, the *Scrubber* performs the first step; when T_{inter} is longer than the estimated time T_{disk} taken by fetching a block from the underlying disk system, the *Scrubber* performs the second step to prefetch $\lfloor T_{inter}/T_{disk} \rfloor$ blocks. Migrated blocks are written to the SSD cache via *Clean Write* interface. In our evaluation, we set m to be 10, T_{crc} equal to the CRC encoding latency and T_{disk} to be the average disk access latency 1.5*ms* [37].

3.5 Garbage Collection

In SSDs, garbage collection (GC) process is executed to reclaim flash space by erasing victim blocks and may bring about significant performance impacts because of its interference with normal activities [16, 37]. The GC overhead mainly comes from consolidating valid pages from victim blocks to clean blocks and erasing victim blocks. *FlexECC* employs a slightly modified greedy algorithm to perform GC. As it is with the normal greedy algorithm [2], *FlexECC* also selects the block which has the highest cleaning efficiency, i.e, the block contains the most invalid pages within it, as the victim block. When migrating the valid pages, CRC-protected pages are relocated to CRC-protected pages, while BCH-encoded pages which have been previously marked out by *Flush Read* are rewritten to elsewhere using *Clean Write* (the "GC write" operation in Figure 2) and the remaining BCH-encoded pages are again relocated to BCH-encoded pages. Due to the discrepancies in decoding latency between CRC-protected pages and BCH-encoded pages, the GC process in *FlexECC* consumes less time than that in conventional SSD. The more CRC-protected pages there are in the victim block, the shorter the GC process would be. The shortened GC process helps improving the overall performance.

3.6 Putting Them All Together

Summarizing the above discussions, we are able to arrive at the conclusion of how a request is handled by *FlexECC*. First, *FlexECC* determines the type (read or write) of the arriving request, and then determines which code scheme is applied to the target page. Reading uncorrupted CRC-protected page is straightforward, while reading corrupted CRC-protected page has to be satisfied by visiting the underlying disk if the page has not been brought in the cache beforehand by the *Scrubber*. A clean write is destinated to a CRC-protected page, and a dirty write is destinated to a BCH-encoded page. Equation 6 and Equation 7 give the estimated read latency T_r and write latency T_w, respectively. In the equations, R_{crc}, R_{bch} and R_{disk} denote the latencies of reading a CRC-protected page, reading a BCH-encoded page and reading a block from disk, respectively. Similarly, W_{crc} and W_{bch} denote the write latencies of writing a CRC-protected page and writing a BCH-encoded page, respectively. ξ is the corruption probability of a flash page which is relevant to the flash memory RBER, software errors, etc. η is the probability of reading a CRC-protected page and γ is the probability of writing a CRC-protected page. The values of η and γ are dependent on the cache manager configuration (e.g., replacement policy, flushing interval, etc.) and the features of workloads.

$$T_r = (1-\xi)(\eta R_{crc} + (1-\eta)R_{bch}) + \xi R_{disk} \qquad (6)$$

$$T_w = \gamma W_{crc} + (1-\gamma)W_{bch} \qquad (7)$$

4 Experimental Evaluation

4.1 Evaluation Methodology

To verify the effectiveness of *FlexECC*, we have implemented a prototype and conducted comprehensive evaluations. The evaluations consist of two steps. First, we add a monitor into the *Flashcache* [37] to track the cache block behaviors. The monitor outputs block access traces having the interface operations defined in Section 3.3. Then we use those traces to drive *FlexECC* which is implemented based on an SSD simulator [2]. We use Filebench [23] to generate four representative workloads, *Fileserver*, *Webserver*, *Mailserver*, *OLTP* and two micro-workloads, *R_75* and *R_90*, both of which are read-intensive workloads having 75% and 90% reads, respectively. Each of the workloads runs for 30 minutes on a *Flashcache*-created hybrid storage system comprising of a 1TB disk as the back-end storage and a 250GB PCI-e SSD as a cache. The cache space and working set sizes are set to 30GB and 360GB, respectively. The write-back cache mode and LRU replacement algorithm are used. Table 1 summarizes the traces characteristics.[4] We assume protection granularity is page-based and the ECC redundancy information is stored in the spare region of each page. Table 2 lists the main relevant operational latencies. We simulate an 8-chip 60GB SSD with 15% overprovisioning space and set ioscale to 100 to slow down replaying those too-intensive traces(they are collected on a PCI-E SSD). In addition, we enable the copy-back operation when performing garbage collection within the SSD.

Table 1: Trace Characteristics

	File	Web	Mail	OLTP	R_75	R_90
READ	15.2%	17.8%	21.9%	8.6%	75%	90%
WRITE	84.8%	82.2%	78.1%	91.4%	25%	10%

Table 2: Operational Latencies

Page Read	$25\mu s$	CRC Encoding	$0.8\mu s$
Page Write	$200\mu s$	CRC Decoding	$0.8\mu s$
Block Erase	1.5ms	BCH Encoding	$0.8\mu s$
BCH Correct Dec.	$5\mu s$	BCH Corrupted Dec.	$10\mu s$

4.2 Performance Comparison

In this section, we report the workloads average request response time to demonstrate the overall performance improvement of *FlexECC* over conventional SSD which is armed with regular BCH schemes. In the figures, these two kinds of devices are denoted as *FlexECC* and *BCH-SSD*, respectively. Figure 3 shows the comparison results. As it is clearly shown in Figure 3, *FlexECC* consistently improves the performance across all the tested workloads relative to the traditional SSD armed with regular BCH scheme. Specifically, *FlexECC* improves performance by 30.2%, 30.1%, 30.8%, 28.5%, 49% and 63.5% for *FileServer*, *MailServer*, *WebServer*, *OLTP*, *R_75* and *R_90*, respectively. Generally, read-intensive workloads benefit more from *FlexECC* than other workloads primarily due to the reduced page decoding latency associated with every CRC-encoded page reading, which is evidenced by the fact that the average read response time has been reduced by averagely around 35% and the shortened garbage collection process, which is further investigated in the next subsection. Unsurprisingly, the write response time exhibits only marginal improvement because the CRC encoding overhead is equal to that of BCH encoding. Figure 4 shows the response time CDF comparisons for workloads *FileServer*, *MailServer*, *WebServer*, *OLTP*. It is observed from the figures that each workload has a certain percentage of requests (covered by the visible red line) that have smaller response time in *FlexECC* than in *BCH-SSD*. Those percentages are nearly equal to the read percentages of the workloads (see Table 1), demonstrating the read requests serving have been speeded up.

It is worth noting that the overall performance improvement is a combined outcome, which means even though we are only able to achieve about 25% performance gain for individual page decoding by replacing BCH with CRC, we have seen more than 30% performance improvement for the workloads. The reason is that the shortened page reading and reduced garbage collection can also alleviate resource contention (e.g., reduce request queuing time) and thus further improves performance.

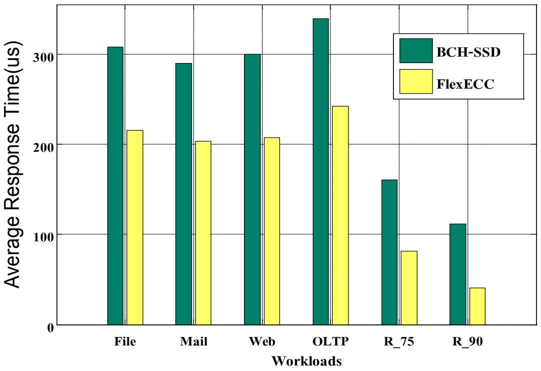
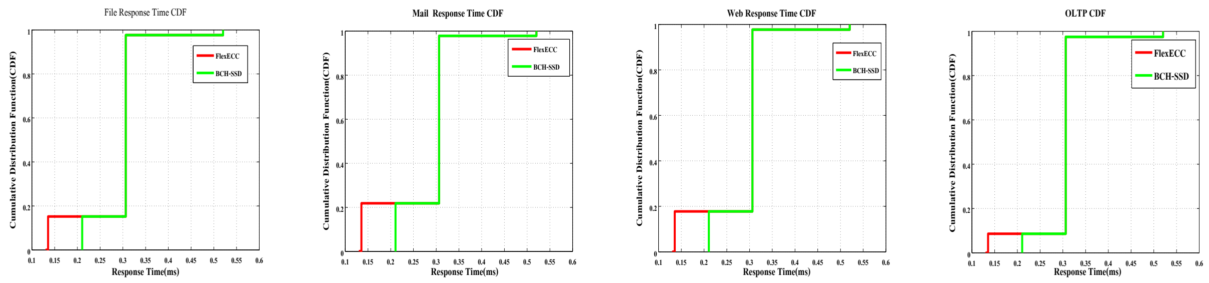


Figure 3: Workloads Average Response Time.

4.3 Garbage Collection

Garbage collection (GC) in SSDs affects performance because it may interfere with ongoing workloads. It spends time in reading out validate pages in victims blocks, writing them to clean blocks and erasing victim blocks. Generally, The shorter time the garbage collection process takes, the less negative impacts it imposes on performance. In our current *FlexECC* implementation, we use the greedy algorithm for selecting victim blocks. The victim blocks may contain CRC-protected pages and BCH-encoded pages. The more CRC-encoded blocks *FlexECC* garbage collects, the faster it migrates valid pages to clean blocks. Figure 5 shows the total cleaning time comparison. From the figure, we notice that compared to *BCH-SSD*, *FlexECC* reduces an impressive amount of total cleaning time, up to 21.8%, 21.8%, 21.7%, 21.7% and 17.8% for *FileServer*, *MailServer*, *WebServer*, *OLTP*, and *R_75*, respectively, even though they have recycled the same number of victim blocks, which are 208829, 215556, 218868, 197455 and 36189, respectively. It is worth noting that the workload *R_90* is not shown in that figure because its total cleaning time is 0. Unlike the average response time, read-intensive workloads do not exhibit the most cleaning time savings because there are fewer page migrations due to the lack of workload writes and associated erasures. Table 3 gives more explanations on the performance gains and garbage collection time reduction, by listing the number of reading BCH-encoded pages (BCH_READ), reading CRC-protected pages (CRC_READ) from the requests and moving CRC-protected pages (CRC_MOVED), transforming BCH-encoded to CRC-protected pages (BCH2CRC) during GC. From the table, we note that read-intensive workloads' performance gains are mainly attributed to the reduced decoding latency rather than saved cleaning time.



(a) File CDF

(b) Mail CDF

(c) Web CDF

(d) OLTP CDF

Figure 4: Workloads Response Time Cumulative Distribution Function (CDF) Comparison.

Table 3: FlexECC Page Statistics

	File	Mail	Web	OLTP	R_75	R_90
BCH_READ	1,517,326	2,420,758	1,899,771	760,480	272,948	8,874,561
CRC_READ	5,191	12,303	6,435	1,265	7,247,563	146,982
CRC_MOVED	3,720,833	3,741,430	3,725,016	3,716,391	1,420,582	0
BCH2CRC	48,501	93,955	57,188	46,688	726	0

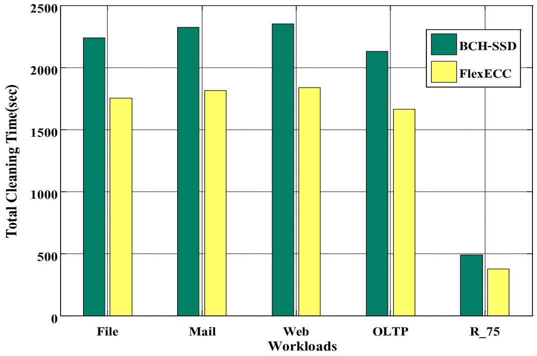


Figure 5: Total Cleaning Time Comparison.

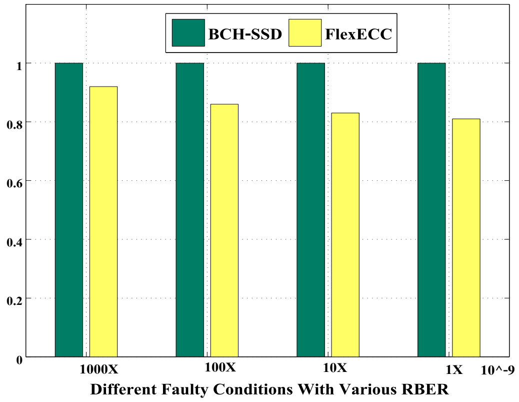


Figure 6: File Performance Under Faulty Conditions.

4.4 Performance Under Faulty Conditions

In this section, we compare the performance under faulty conditions. For simplicity but without loss of generality, we introduce errors to flash pages according to specific raw bit error rates (RBER). In more detail, given a specific RBER, we assume that the every $\frac{1}{RBER}$th bit is corrupted and the page contains this corrupted bit is considered erroneous. Corruptions could occur to both CRC-pages and BCH-pages. We also assume faulty pages only impact reads, because writes are inherently indirected at the FTL layer and thus bypass faulty pages. If the corrupted pages is CRC-protected, then reading it must be serviced by accessing the underlying disk, otherwise it is assumed to be corrected via BCH decoding.

Figure 6 shows the normalized average response time relative to *BCH-SSD* without errors for *Fileserver* workload. As demonstrated in the figure, even under faulty conditions, *FlexECC* still outperforms *BCH-SSD* in error-free conditions, achieving an average 20% improvement. This is because the performance gains brought by partially replacing BCH-encoded with CRC-protected pages and the *Scrubber* prefetching dwarf the overhead associated with handling accesses to corrupted pages. The statistics in Table 4 gives an in-depth explanation regarding the behind reasons. It lists the statistics for *FileServer*, *R_75* and *R_90* workloads. We can make the following observations. First, for write-intensive workloads (i.e., *Fileserver*) there is almost no disk accesses. That's because write requests postpone the visits to corrupted pages and thus increase their probability of being prefetched by *Scrubber*. For read-intensive

Table 4: Corruption Related Statistics When RBER=10^{-7}

Workloads	File	R_75	R_90	Notes
Corruptions	4107	4107	4107	# of corruptions introduced during running
Disk Access	0	8	7	# of disk accesses
Prefetched	2720	3917	4024	# of corrupted pages prefetched by *scrubber*
BCH Decoded	195	34	10	# of corrected pages via BCH decoding

workloads, there are disk accesses happening, because the corrupted pages would be visited with a high probability. Second, the number of disk access is rather small, because most corrupted pages have been prefetched in advance, which illustrates the *Scrubber* is efficient in leveraging workload idleness. Third, the high numbers of prefetched pages of *R_75* and *R_90* are attributed to the fact that they contain a high percentage of CRC-pages, which is also evidenced by the dominance of CRC_READ and CRC_MOVED in Table 3. It should be noted that the sum of *Disk Access*, *Prefetched* and *BCH Decoded* is not equal to *Corruptions*, since there could be corrupted pages that have not yet been prefetched or corrected.

5 Related Work

Flash-based SSDs have been extensively researched as a cache due to their widespread deployment in HDD-SSD hybrid storage systems. Kgil et al. [21] propose to partition the NAND flash cache space into read and write caches and employ a programmable flash memory controller to improve performance and reliability. They also utilize CRC within the cache device, but in a complementary way to reduce BCH's false positives, as opposed to our replacement of BCH for clean pages. Yang et al. [45] propose to improve SSD cache endurance via reducing media writes and erases. Koller et al. [22] present a study discussing write policies and consistency problems of SSD cache deployed in networked environment. More recently, Holland et al. [18] explore the design space of flash as a cache in the storage client side instead of server side and make several interesting findings. Albrecht et al. [4] present Janus, a cloud-scale distributed file system that is actively-used in Google Inc. In their paper, they formulate and solve an optimization problem to determine the flash cache allocation to workloads according to their respective *cacheability* and conclude that flash storage is a cost-effective complement to disks in data centers. These works all use flash-based SSD as a cache without taking into account synergistic optimizations. Our proposed *FlexECC* expands the design space from a new dimension and could be integrated into these systems to further improve the cache performance.

As flash technology scales, the reliability issue associated with increasing flash memory bit error rate and the required error correction code have specially received research interests. Mielke et al. [29] conduct a comprehensive study of bit error rate of MLC SSDs from different manufacturers. Grupp et al. [15] observe a trend of decreasing performance and reliability. Observing ECC is under-utilized most of the time, especially when SSDs are in their early usage stage, Pan et al. [33] propose to speed up writes and tolerate more defective cells by fully exploiting ECC's capability. Taking advantage of the retention time gap between specification and actual requirements, Liu el al. [25] propose to improve write performance and/or reduce ECC overhead by relaxing retention time. Wu et al. [44] propose to adaptively use different ECCs according to workloads to avoid consistently using strong ECC. Similarly, Cai et al. [6] suggest a technique called *Correct and Refresh* to avoid using strong ECC. Their idea is to periodically refresh charges in memory cells to reduce the dominant retention errors due to loss of charges. The prolonged ECC decoding latency problem associated with advanced ECC schemes in modern SSDs has recently been observed by Zhao [49]. In their work, they suggest effective methods to reduce the decoding latency of LDPC codes. While each of these works tries to make a preferential trade-off toward performance, reliability, or cost when designing ECC schemes for flash memory, our work differs from them in that *FlexECC* is cache-oriented and can safely get rid of ECC for clean flash pages.

Relaxing ECC for performance and/or energy purposes has also been explored in the memory systems. Yoon et al. [46, 47] suggest a two-tiered error protection mechanism for last-level cache. The tier-1 code is located with the protected cache line and only provides error detection, while the tier-2 code is stored on off-chip DRAM. In this scheme, the ECC consumes limited on-chip SRAM resource, but is able to provide arbitrarily strong tier-2 protection. Based on the observations that in low-power operating mode different cache lines exhibit different reliability characteristics, Alameldeen et al. [3] propose a variable-strength ECC scheme, in which cache lines having zero or single error are protected by fast and simple ECC, while cache lines having multiple errors are protected by stronger ECC.

The most relevant work to *FlexECC* is SSC [37] in that SSC also proposes a cache-oriented SSD architecture and extends interfaces between applications and the device. However, the performance improvement of SSC mainly comes from the elimination of page migrations during garbage collection, while *FlexECC* benefits from reduced decoding latency. A potential shortcoming of SSC is that it might exhibit a high miss rate if the silently evicted pages are requested again in the future. By contrast, *FlexECC* only directs accesses to corrupted blocks to beneath storage system. Moreover, in *FlexECC*, every page read benefits from the reduced decoding latency. It should be interesting to quantitatively compare *FlexECC* with SSC, as planned in our future work.

6 Conclusions and Future Work

This paper presents *FlexECC*, a novel high-performance cache-oriented MLC SSD. It flexibly applies BCH or CRC to incoming page writes according to their storage requirement information which is conveyed down by an upper-level cache manager. We have given a theoretic analysis on the decoding latency of BCH and CRC and found the gap in their decoding latencies. Experimental results with a variety of workloads have shown that *FlexECC* is capable of improving the overall cache performance by an impressive extent and save the total amount of cleaning time, without compromising reliability and consistency. As part of the future work, we plan to further improve *FlexECC*'s cache performance by leveraging the space that is otherwise consumed by ECC redundant information as additional effective cache space. In addition, we also plan to investigate the energy savings by replacing off-the-shelf SSD caches with *FlexECC*, especially in a cloud environment. We believe in light of the trend of increasing flash memory RBER and widespread use of MLC SSD as caches, it could be significantly beneficial to deploy *FlexECC* in practical systems.

Acknowledgment

We would like to thank the anonymous reviewers for their valuable feedbacks and constructive suggestions. This research is partially supported by the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1102605, CCF-1102624, and CNS-1218960, and the National Basic Research Program (973 Program) of China under Grant No.2011CB302305, the National Natural Science Foundation of China under Grant No. 61232004. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] Introduction to EMC VFCache. White paper. http://www.emc.com/collateral/hardware/white-papers/h10502-vfcache-intro-wp.pdf, February 2012.

[2] Agrawal, N., Prabhakaran, V., Wobber, T., and Davis, J. D. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX ATC* (2008).

[3] Alameldeen, A. R., Wagner, I., Chishti, Z., Wu, W., Wilkerson, C., and Lu, S.-L. Energy-Efficient Cache Design Using Variable-Strength Error-Correcting Codes. In *Proceedings of ISCA* (2011).

[4] Albrecht, C., Merchant, A., Stokely, M., Waliji, M., Labelle, F., Coehlo, N., Shi, X., and Schrock, C. E. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Proceedings of the 2013 USENIX Annual Technical Conference(USENIX ATC)* (2013).

[5] Badam, A., and Pai, V. S. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of NSDI* (2011).

[6] Cai, Y., Yalcin, G., Mutlu, O., Haratsch, E. F., Cristal, A., Unsal, O. S., and Mai, K. Flash Correct-and-Refresh: Retention-Aware Error Management for Increased Flash Memory Lifetime. In *Proceedings of the* 30th *IEEE International Conference on Computer Design(ICCD)* (2012).

[7] Chien, A. A., and Karamcheti, V. Moore's Law: The First Ending and a New Beginning. *IEEE Computer Magazine 46*, 12 (December 2013), 48–53.

[8] Cho, J., and Sung, W. Efficient Software-Based Encoding and Decoding of BCH Codes. *IEEE Transations on Computers 58*, 7 (July 2009), 878–889.

[9] Choi, H., Liu, W., and Sung, W. VLSI Implementation of BCH Error Correction for Multilevel Cell NAND Flash Memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 18*, 5 (April 2010), 843–847.

[10] C.Zambelli, M.Indaco, M.Fabiano, Carlo, S., P.Prinetto, P.Olivo, and D.Bertozzi. A Cross-Layer Approach for New Reliability-Performance Trade-Offs in MLC NAND Flash Memories. In *Proceedings of the Design Automation & Test in Europe Conference & Exhibition(DATE)* (2012).

[11] Deal, E. Trends in NAND Flash Memory Error Correction. *Cyclic Design White Paper* (2009).

[12] Deng, Y., Lu, L., Zou, Q., Huang, S., and Zhou, J. Modeling the Aging Process of Flash Storage by Leveraging Semantic I/O. *Future Generation Comp. Syst. 32* (March 2014), 338–344.

[13] Deng, Y., and Zhou, J. Architectures and Optimization Methods of Flash Memory Based Storage Systems. *Journal of Systems Architecture 57*, 2 (February 2011), 214–227.

[14] Grupp, L. M., Caulfield, A. M., Coburn, J., and Swanson, S. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *Proceedings of MICRO* (2009).

[15] Grupp, L. M., Davis, J. D., and Swanson, S. The Bleak Future of NAND Flash Memory. In *Proceedings of the* 10th *USENIX Conference on File and Storage Technologies(FAST)* (2012).

[16] Grupp, L. M., Davis, J. D., and Swanson, S. The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference(USENIX)* (2013).

[17] Gunawi, H. S., Prabhakaran, V., Krishnan, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Improving File System Reliability with I/O Shepherding. In *Proceedings of SOSP* (2007).

[18] Holland, D. A., Angelino, E., Wald, G., and Seltzer, M. I. Flash Caching on the Storage Client. In *Proceedings of the USENIX ATC* (2013).

[19] Huang, P., Wu, G., He, X., and Xiao, W. An Aggressive Worn-out Flash Block Management Scheme to Alleviate SSD Performance Degradation. In *Proceedings of Eurosys* (2014).

[20] Huang, P., Zhou, K., Wang, H., and Li, C. BVSSD: Build Built-in Versioning Flash-based Solid State Drives. In *Proceedings of SYSTOR* (2012).

[21] Kgil, T., Roberts, D., and Mudge, T. Improving NAND Flash Based Disk Caches. In *Proceedings of the* 35^{th} *Annual International Symposium on Computer Architecture(ISCA)* (2008).

[22] Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N., and Zhao, M. Write Policies for Host-side Flash Caches. In *Proceedings of the* 11^{th} *USENIX Conference on File and Storage Technologies (FAST)* (2013).

[23] Lee, E., Bahn, H., , and Noh, S. H. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of FAST* (2013).

[24] Lin, S., and Costello, D. J. *Error Control Coding(2nd Edition)*. Prentice Hall,Inc., 2004.

[25] Liu, R.-S., Yang, C.-L., and Wu, W. Optimizing NAND Flash-Based SSDs via Retention Relaxation. In *Proceedings of FAST* (2012).

[26] Lu, Y., Shu, J., and Zheng, W. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Proceedings of FAST* (2013).

[27] Luo, T., Ma, S., Lee, R., Zhang, X., Liu, D., and Zhou, L. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. In *Proceedings of the* 22^{nd} *International Conference on Parallel Architectures and Compilation Techniques(PACT)* (2013).

[28] Mesnier, M., Chen, F., Luo, T., and Akers, J. B. Differentiated Storage Services. In *Proceedings of the* 23^{rd} *ACM Symposium on Operating Systems Principles(SOSP)* (2011).

[29] Mielke, N., Marquart, T., Wu, N., Kessenich, J., , Belgal, H., Schares, E., and Trivedi, F. Bit Error Rate in NAND Flash Memories. In *Proceedings of IEEE International Reliability Physics Symposium(IRPS)* (2008).

[30] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proceedings of the* 4^{th} *ACM European conference on Computer systems(Eurosys)* (2009).

[31] Oprea, A., and Juels, A. A Clean-Slate Look at Disk Scrubbing. In *Proceedings of the* 8^{th} *USENIX conference on File and storage technologies(FAST)* (2010).

[32] Pan, Y., Dong, G., Wu, Q., and Zhang, T. Quasi-Nonvolatile SSD: Trading Flash Memory Nonvolatility to Improve Storage System Performance for Enterprise Applications. In *Proceedings of the* 18^{th} *IEEE International Symposium on High Performance Computer Architecture(HPCA)* (2012).

[33] Pan, Y., Dong, G., and Zhang, T. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *Proceedings of the* 9^{th} *USENIX Conference on File and Stroage Technologies(FAST)* (2011).

[34] Park, S., and Shen, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedings of FAST* (2012).

[35] Pritchett, T., and Thottethodi, M. SieveStore: A Highly-Selective, Ensemble-level Disk Cache for Cost-Performance. In *Proceedings of ISCA* (2010).

[36] Ren, J., and Yang, Q. I-CASH: Intelligently Coupled Array of SSDs and HDDs. In *Proceedings of HPCA* (2011).

[37] Saxena, M., Swift, M. M., and Zhang, Y. FlashTier: a Lightweight, Consistent and Durable Storage Cache. In *Proceedings of Eurosys* (2012).

[38] Saxena, M., Zhang, Y., Swift, M. M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of FAST* (2013).

[39] Shen, K., and Park, S. FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference(USENIX ATC)* (2013).

[40] Strukov, D. The Area and Latency Tradeoffs of Binary Bit-parallel BCH Decoders for Prospective Nanoelectronic Memories. In *Proceedings of Fortieth Asilomar Conference on Signals, Systems and Computers(ACSSC)* (2006).

[41] Sun, F., Rose, K., and Zhang, T. On the Use of Strong BCH Codes for Improving Multilevel NAND Flash Memory Storage Capacity. In *Proceedings of the IEEE Workshop on Signal Processing Systems(SiPS): Design and Implementation* (2006).

[42] Ungureanu, C., Debnath, B., Rago, S., and Aranya, A. TBF: A Memory-Efficient Replacement Policy for Flash-based Caches. In *Proceedings of the IEEE* 29^{th} *International Conference on Data Engineering (ICDE)* (2013).

[43] Wu, G., and He, X. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *Proceedings of the* 10^{th} *USENIX conference on File and Storage Technologies(FAST)* (2012).

[44] Wu, G., He, X., Xie, N., and Zhang, T. DiffECC: Improving SSD Read Performance Using Differentiated Error Correction Coding Schemes. In *Proceedings of the* 18^{th} *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems(MASCOTS)* (2010).

[45] Yang, J., Plasson, N., Gillis, G., Talagala, N., Sundararaman, S., and Wood, R. HEC: Improving Endurance of High Performance Flash-based Cache Devices. In *Proceedings of the* 6^{th} *Annual International Systems and Storage Conference(SYSTOR)* (2013).

[46] Yoon, D. H., and Erez, M. Flexible Cache Error Protection using an ECC FIFO. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis(SC)* (2009).

[47] Yoon, D. H., and Erez, M. Memory Mapped ECC: Low-Cost Error Protection for Last Level Caches. In *Proceedings of the* 36^{th} *Annual International Symposium on Computer Architecture(ISCA)* (2009).

[48] Zhang, Y., Soundararajan, G., Storer, M. W., Bairavasundaram, L. N., Subbiah, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Warming up Storage-Level Caches with Bonfire. In *Proceedings of FAST* (2013).

[49] Zhao, K., Zhao, W., Sun, H., Zhang, T., Zhang, X., and Zheng, N. LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. In *Proceedings of FAST* (2013).

Notes

[1] In context of cache, single bit detection capability of EDC can typically fulfill the purpose, so our analyzed speedup is conservative.

[2] We have investigated $T_m = \lambda T_a$ with varying λ ($\lambda > 1$) as well and have reached the similar conclusion.

[3] In reality, it is more common to use a block as the granularity. For simplicity, we assume page granularity.

[4] The characteristics are different from [23], because *Flashcache* has bypassed non-4KB and large (more than 128KB) sequential requests directly to HDD. All trace requests are 4KB in size.

Nitro: A Capacity-Optimized SSD Cache for Primary Storage

Cheng Li[†], Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace
[†]*Rutgers University* *EMC Corporation – Data Protection and Availability Division*

Abstract

For many primary storage customers, storage must balance the requirements for large capacity, high performance, and low cost. A well studied technique is to place a solid state drive (SSD) cache in front of hard disk drive (HDD) storage, which can achieve much of the performance benefit of SSDs and the cost per gigabyte efficiency of HDDs. To further lower the cost of SSD caches and increase effective capacity, we propose the addition of data reduction techniques.

Our cache architecture, called Nitro, has three main contributions: (1) an SSD cache design with adjustable deduplication, compression, and large replacement units, (2) an evaluation of the trade-offs between data reduction, RAM requirements, SSD writes (reduced up to 53%, which improves lifespan), and storage performance, and (3) acceleration of two prototype storage systems with an increase in IOPS (up to 120%) and reduction of read response time (up to 55%) compared to an SSD cache without Nitro. Additional benefits of Nitro include improved random read performance, faster snapshot restore, and reduced writes to SSDs.

1 Introduction

IT administrators have struggled with the complexity, cost, and overheads of a primary storage architecture as performance and capacity requirements continue to grow. While high IOPS, high throughput, and low latency are necessary for primary workloads, many customers have budget limitations. Therefore, they also want to maximize capacity and management simplicity for a given investment. Balancing these requirements is an ongoing area of storage research.

Fundamentally though, the goals of high performance and cost-efficient storage are in conflict. Solid state drives (SSDs) can support high IOPS with low latency, but their cost will be higher than hard disk drives (HDDs) for the foreseeable future [24]. In contrast, HDDs have high capacity at relatively low cost, but IOPS and latency are limited by the mechanical movements of the drive. Previous work has explored SSDs as a cache in front of HDDs to address performance concerns [1, 7, 30], and the SSD interface has been modified for caching purposes [26], but the cost of SSDs continues to be a large fraction of total storage cost.

Our solution, called Nitro, applies advanced data reduction techniques to SSD caches, increasing the effective cache size and reducing SSD costs for a given system. Deduplication (replacing a repeated data block with a reference) and compression (e.g. LZ) of storage have become the primary strategies to achieve high space and energy efficiency, with most research performed on HDD systems. We refer to the combination of deduplication and compression for storage as capacity-optimized storage (COS), which we contrast with traditional primary storage (TPS) without such features.

Though deduplicating SSDs [4, 13] and compressing SSDs [10, 19, 31] has been studied independently, using both techniques in combination for caching introduces new complexities. Unlike the variable-sized output of compression, the Flash Translation Layer (FTL) supports page reads (e.g. 8KB). The multiple references introduced with deduplication conflicts with SSD erasures that take place at the block level (a group of pages, e.g. 2MB), because individual pages of data may be referenced while the rest of a block could otherwise be reclaimed. Given the high churn of a cache and the limited erase cycles of SSDs, our technique must balance performance concerns with the limited lifespan of SSDs. We believe this is the first study combining deduplication and compression to achieve capacity-optimized SSDs.

Our design is motivated by an analysis of deduplication patterns of primary storage traces and properties of local compression. Primary storage workloads vary in how frequently similar content is accessed, and we wish to minimize deduplication overheads such as in-memory indices. For example, related virtual machines (VMs) have high deduplication whereas database logs tend to have lower deduplication, so Nitro supports targeting deduplication where it can have the most benefit. Since compression creates variable-length data, Nitro packs compressed data into larger units, called Write-Evict Units (WEUs), which align with SSD internal blocks. To extend SSD lifespan, we chose a cache replacement policy that tracks the status of WEUs instead of compressed data, which reduces SSD erasures. An important finding is that replacing WEUs instead of small data blocks maintains nearly the same cache hit ratio and performance of finer-grained replacement, while extending SSD lifespan.

To evaluate Nitro, we developed and validated a simulator and two prototypes. The prototypes place Nitro in front of commercially available storage products. The first prototype uses a COS system with deduplication and compression. The system is typically targeted for storing highly redundant, sequential backups. Therefore, it has lower random I/O performance, but it becomes a plausible primary storage system with Nitro acceleration. The second prototype uses a TPS system without deduplication or compression, which Nitro also accelerates.

Because of the limited computational power and memory of SSDs [13] and to facilitate the use of off-the-shelf SSDs, our prototype implements deduplication and compression in a layer above the SSD FTL. Our evaluation demonstrates that Nitro improves I/O performance because it can service a large fraction of read requests from an SSD cache with low overheads. It also illustrates the trade-offs between performance, RAM, and SSD lifespan. Experiments with prototype systems demonstrate additional benefits including improved random read performance in aged systems, faster snapshot restore when snapshots overlap with primary versions in a cache, and reduced writes to SSDs because of duplicate content. In summary, our contributions are:

- We propose Nitro, an SSD cache that utilizes deduplication, compression, and large replacement units to accelerate primary I/O.
- We investigate the trade-offs between deduplication, compression, RAM requirements, performance, and SSD lifespan.
- We experiment with both COS and TPS prototypes to validate Nitro's performance improvements.

2 Background and Discussion

In this section, we discuss the potential benefits of adding deduplication and compression to an SSD cache and then discuss the appropriate storage layer to add a cache.

Leveraging duplicate content in a cache. I/O rates for primary storage can be accelerated if data regions with different addresses but duplicate content can be reused in a cache. While previous work focused on memory caching and replicating commonly used data to minimize disk seek times [14], we focus on SSD caching.

We analyzed storage traces (described in §5) to understand opportunities to identify repeated content. Figure 1 shows the deduplication ratio (defined in §5.1) for 4KB blocks for various cache sizes. The deduplication ratios increase slowly for small caches and then grow rapidly to $\sim 2.0X$ when the cache is sufficiently large to hold the working set of unique content. This result confirms that a cache has the potential to capture a significant fraction of potential deduplication [13].

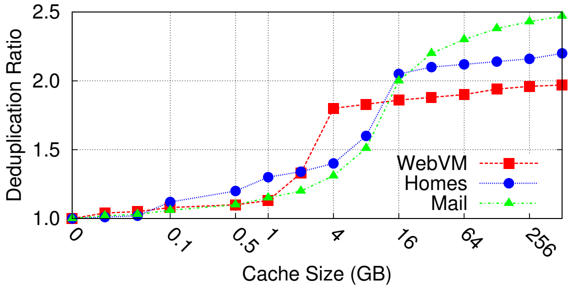


Figure 1: Caching tens of thousand of blocks will achieve most of the potential deduplication.

This result motivates our efforts to build a deduplicated SSD cache to accelerate primary storage. Adding deduplication to a storage system increases complexity, though, since infrastructure is needed to track the liveness of blocks. In contrast, caching requires less complexity, since cache misses do not cause a data loss for write-through caching, though performance is affected. Also, the overhead of calculating and managing secure fingerprints must not degrade overall performance.

Leveraging compression in a cache. Compression, like deduplication, has the potential to increase cache capacity. Previous studies [5, 10, 29, 31] have shown that local compression saves from 10-60% of capacity, with an approximate mean of 50% using a fast compressor such as LZ. Potentially doubling our cache size is desirable, as long as compression and decompression overheads do not significantly increase latency. Using an LZ-style compressor is promising for a cache, as compared to a HDD system that might use a slower compressor that achieves higher compression. Decompression speed is also critical to achieve low latency storage, so we compress individual data blocks instead of concatenating multiple data blocks before compression. Our implementation has multiple compression/decompression threads, which can leverage future advances in multicore systems.

A complexity of using compression is that it transforms fixed-sized blocks into variable-sized blocks, which is at odds with the properties of SSDs. Similar to previous work [10, 19, 31], we pack compressed data together into larger units (WEUs). Our contribution focuses on exploring the caching impact of these large units, which achieves compression benefits while decreasing SSD erasures.

Appropriate storage layer for Nitro. Caches have been added at nearly every layer of storage systems: from client-side caches to the server-side, and from the protocol layer (e.g. NFS) down to caching within hard drives. For a deduplicated and compressed cache, we believe there are two main locations for a server-side cache. The first is at the highest layer of the storage stack, right after processing the storage protocol. This is the server's first opportunity to cache data, and it is as close to the client as possible, which minimizes latency.

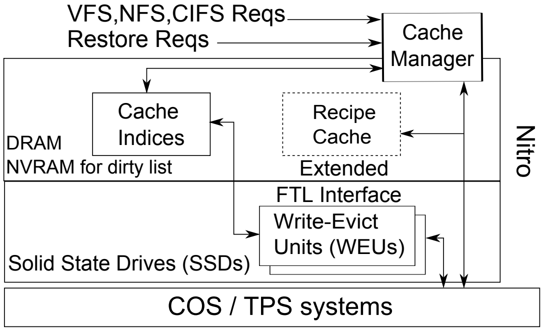


Figure 2: SSD cache and disk storage.

The second location to consider is post-deduplication (and compression) within the system. The advantage of the post-deduplication layer is that currently existing functionality can be reused. Of course, deduplication and compression have not yet achieved wide-spread implementation in storage systems. An issue with adding a cache at the post-deduplication layer is that some mechanism must provide the file recipe, a structure mapping from file and offset to fingerprint (e.g. SHA-1 hash of data), for every cache read. Loading file recipes adds additional I/O and latency to the system, depending on the implementation. While we added Nitro at the protocol layer, for COS systems, we evaluate the impact of using file recipes to accelerate duplicate reads (§3.1). We then compare to TPS systems that do not typically have file recipes, but do benefit from caching at the protocol layer.

3 Nitro Architecture

This section presents the design of our Nitro architecture. Starting at the bottom of Figure 2, we use either COS or TPS HDD systems for large capacity. The middle of the figure shows SSDs used to accelerate performance, and the upper layer shows in-memory structures for managing the SSD and memory caches.

Nitro is conceptually divided into two halves shown in Figure 2 and in more detail in Figure 3 (steps 1-6 are described in §3.2). The top half is called the *CacheManager*, which manages the cache infrastructure (indices), and a lower half that implements SSD caching. The CacheManager maintains a file index that maps the file system interface (<filehandle, offset>) to internal SSD locations; a fingerprint index that detects duplicate content before it is written to SSD; and a dirty list that tracks dirty data for write-back mode. While our description focuses on file systems, other storage abstractions such as volumes or devices are supported. The CacheManager is the same for our simulator and prototype implementations, while the layers below differ to either use simulated or physical SSDs and HDDs (§4).

We place a small amount of NVRAM in front of our cache to buffer pending writes and to support write-back caching: check-pointing and journaling of the dirty list. The CacheManager implements a dynamic prefetching scheme that detects sequential accesses when the consecutive bytes accessed metric (M11 in [17]) is higher than a threshold across multiple-streams. Our cache is scan-resistant because prefetched data that is accessed only once in memory will not be cached. We currently do not cache file system metadata because we do not expect it to deduplicate or compress well, and we leave further analysis to future work.

3.1 Nitro Components

Extent. An extent is the basic unit of data from a file that is stored in the cache, and the cache indices reference extents that are compressed and stored in the SSDs. We performed a large number of experiments to size our extents, and there are trade-offs in terms of read-hit ratio, SSD erasures, deduplication ratio, and RAM overheads. As one example, smaller extents capture finer-grained changes, which typically results in higher deduplication ratios, but smaller extents require more RAM to index. We use the median I/O size of the traces we studied (8KB) as the default extent size. For workloads that have differing deduplication and I/O patterns than what we have studied, a different extent size (or dynamic sizing) may be more appropriate.

Write-Evict Unit (WEU). The Write-Evict Unit is our unit of replacement (writing and evicting) for SSD. File extents are compressed and packed together into one WEU in RAM, which is written to an SSD when it is full. Extents never span WEUs. We set the WEU size equal to one or multiple SSD block(s) (the unit for SSD erase operation) depending on internal SSD properties, to maximize parallelism and reduce internal fragmentation. We store multiple file extents in a WEU. Each WEU has a header section describing its contents, which is used to accelerate rebuilding the RAM indices at start-up. The granularity of cache replacement is an entire WEU, thus eliminating copy forward of live-data to other physical blocks during SSD garbage collection (GC). This replacement strategy has the property of reducing erasures within an SSD, but this decision impacts performance, as we discuss extensively in §6.1. WEUs have generation numbers indicating how often they have been replaced, which are used for consistency checks as described later.

File index. The file index contains a mapping from filehandle and offset to an extent's location in a WEU. The location consists of the WEU ID number, the offset within the WEU, and the amount of compressed data to read. Multiple file index entries may reference the same extent due to deduplication. Entries may also be marked as dirty if write-back mode is supported (shown in gray in Figure 3).

Fingerprint index. To implement deduplication within the SSDs, we use a fingerprint index that maps from ex-

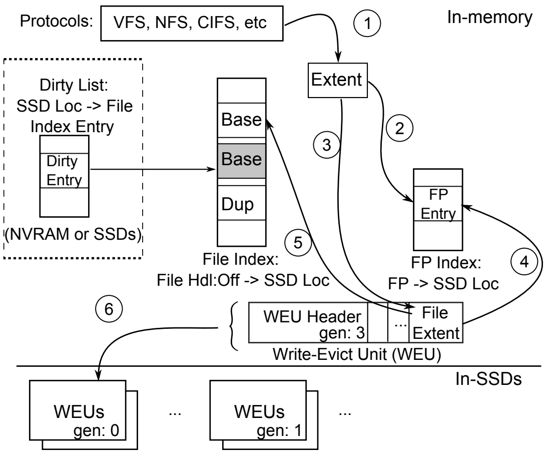


Figure 3: File index with base and duplicate entries, fingerprint index, file extents stored in WEUs, and a dirty list.

tent fingerprint to an extent's location within the SSD. The fingerprint index allows us to find duplicate entries and effectively increase the cache capacity. Since primary workloads may have a wide range of content redundancy, the fingerprint index size can be limited to any arbitrary level, which allows us to make trade-offs between RAM requirements and how much potential deduplication is discovered. We refer to this as the *fingerprint index ratio*, which creates a partial fingerprint index. For a partial fingerprint index, a policy is needed to decide which extents should be inserted into/evicted from the fingerprint index. User-specified configurations, folder/file properties, or access patterns could be used in future work. We currently use LRU eviction, which performed as well as more complicated policies.

Recipe cache. To reduce misses on the read-path, we create a cache of file recipes (Figure 2), which represent a file as a sequence of fingerprints referencing extents. This allows us to check the fingerprint index for already cached, duplicate extents. File recipes are a standard component of COS systems and can be prefetched to our cache, though this requires support from the COS system. Since fingerprints are small (20 bytes) relative to the extent size (KBs), prefetching large lists of fingerprints in the background can be efficient compared to reading the corresponding data from HDD storage. A recipe cache can be an add-on for TPS to opportunistically improve read performance. We do not include a recipe cache in our current TPS implementation because we want to isolate the impact of Nitro without changing other properties of the underlying systems. Its impact on performance is discussed in §6.2.

Dirty list. The CacheManager supports both write-through and write-back mode. Write-through mode assumes all data in the cache are clean because writes to the system are acknowledged when they are stable both in SSD and the underlying storage system. In contrast, write-back mode treats writes as complete when data are cached either in the NVRAM or SSD. In write-back mode, a dirty list tracks dirty extents, which have not yet propagated to the underlying disk system. The dirty list can be maintained in NVRAM (or SSD) for consistent logging since it is a compact list of extent locations. Dirty extents are written to the underlying storage system either when they are evicted from the SSD or when the dirty list reaches a size watermark. When a dirty file index entry is evicted (base or duplicate), the file recipe is also updated. The CacheManager then marks the corresponding file index entries as clean and removes the dirty list entries.

3.2 Nitro Functionality

File read path. Read requests check the file index based on filehandle and offset. If there is a hit in the file index, the CacheManager will read the compressed extent from a WEU and decompress it. The LRU status for the WEU is updated accordingly. For base entries found in the file index, reading the extent's header from SSD can confirm the validity of the extent. When reading a duplicate entry, the CacheManager confirms the validity with WEU generation numbers. An auxiliary structure tracks whether each WEU is currently in memory or in SSD.

If there is a file index miss and the underlying storage system supports file recipes (i.e. COS), the CacheManager prefeteches the file recipe into the recipe cache. Subsequent read requests reference the recipe cache to access fingerprints, which are checked against the cache fingerprint index. If the fingerprint is found to be a duplicate, then cached data can be returned, thus avoiding a substantial fraction of potential disk accesses. The CacheManager updates the LRU status for the fingerprint index if there is a hit. If a read request misses in both the file and fingerprint indices, then the read is serviced from the underlying HDD system, returned to the client, and passed to the cache insertion path.

File write path. On a write, extents are buffered in NVRAM and passed to the CacheManager for asynchronous SSD caching.

Cache insertion path. To demonstrate the process of inserting an extent into the cache and deduplication, consider the following 6-step walk-through example in Figure 3: (1) Hash a new extent (either from caching a read miss or from the file write path) to create a fingerprint. (2) Check the fingerprint against the fingerprint index. If the fingerprint is in the index, update the appropriate LRU status and go to step 5. Otherwise continue with step 3. (3) Compress and append the extent to a WEU that is in-memory, and update the WEU header. (4) Update the fingerprint index to map from a fingerprint to WEU location. (5) Update the file index to map from file handle and offset to WEU. The first entry for the cached extent is marked as a “Base” entry. Note that the WEU header only tracks the base entry. (6) When an in-memory WEU becomes full, increment the generation number and write it to the SSD. In write-back mode, dirty extents and clean extents are segregated into separate WEUs to simplify eviction, and the dirty-list is updated when a WEU is migrated to SSD.

Handling of duplicate entries is slightly more complicated. Once a WEU is stored in SSD, we do not update its header because of the erase penalty involved. When a write consists of duplicate content, as determined by the fingerprint index, a duplicate entry is created in the file index (marked as “Dup”) which points to the extent’s location in SSD WEU. Note that when a file extent is over-written, the file index entry is updated to refer to the newest version. Previous version(s) in the SSD may still be referenced by duplicate entries in the file index.

SSD cache replacement policy. Our cache replacement policy selects a WEU from the SSD to evict before reusing that space for a newly packed WEU. The CacheManager initiates cache replacement by migrating dirty data from the selected WEU to disk storage and removing corresponding invalid entries from the file and fingerprint indices. To understand the interaction of WEU and SSDs, we experimented with moving the cache replacement decisions to the SSD, on the premise that the SSD FTL has more internal knowledge. In our co-designed SSD version (§4), the CacheManager will query the SSD to determine which WEU should be replaced based on recency. If the WEU contains dirty data, the CacheManager will read the WEU and write dirty extents to underlying disk storage.

Cleaning the file index. When evicting a WEU from SSD, our in-memory indices must also be updated. The WEU metadata allows us to remove many file index entries. It is impractical, though, to record back pointers for all duplicate entries in the SSD, because these duplicates may be read/written hours or days after the extent is first written to a WEU. Updating a WEU header with a back pointer would increase SSD churn. Instead, we use asynchronous cleaning to remove invalid, duplicate file index entries. A background cleaning thread checks all duplicate entries and determines whether their generation number matches the WEU generation number. If a stale entry is accessed by a client before it is cleaned, then a generation number mismatch indicates that the entry can be removed. All of the WEU generation numbers can be kept in memory, so these checks are quick, and rollover cases are handled.

Faster snapshot restore/access. Nitro not only accelerates random I/Os but also enables faster restore and/or access of snapshots. The SSD can cache snapshot data as well as primary data for COS storage, distinguished by separate snapshot file handles.

We use the standard snapshot functionality of the storage system in combination with file recipes for COS. When reading a snapshot, its recipe will be prefetched from disk into a recipe cache. Using the fingerprint index, duplicate reads will access extents already in the cache, so any shared extents between the primary and snapshot versions can be reused, without additional disk I/O. To accelerate snapshot restores for TPS, integration with differential snapshot tracking is needed.

System restart. Our cache contains numerous extents used to accelerate I/O, and warming up a cache after a system outage (planned or unplanned) could take many hours. To accelerate cache warming, we implemented a system restart/crash recovery technique [26]. A journal tracks the dirty and invalid status of extents. When recovering from a crash, the CacheManager reads the journal, the WEU headers from SSD (faster than reading all extent headers), and recreates indices. Note that our restart algorithm only handles base entries and duplicate entries that reference dirty extents (in write-back mode). Duplicate entries for clean extents are not explicitly referenced from WEU headers, but they can be recovered efficiently by fingerprint lookup when accessed by a client, with only minimal disk I/O to load file recipes.

4 Nitro Implementation

To evaluate Nitro, we developed a simulator and two prototypes. The CacheManager is shared between implementations, while the storage components differ. Our simulator measures read-hit ratios and SSD churn, and its disk stub generates synthetic content based on fingerprint. Our prototypes measure performance and use real SSDs and HDDs.

Potential SSD customization. Most of our experiments use standard SSDs without any modifications, but it is important to validate our design choices against alternatives that modify SSD functionality. Previous projects [1, 3, 13] showed that the design space of the FTL can lead to diverse SSD characteristics, so we would like to understand how Nitro would be affected by potential SSD changes. Interestingly, we found through simulation that Nitro performs nearly as well with a commercial SSD as with a customized SSD.

We explored two FTL modifications, as well as changes to the standard GC algorithm. First, the FTL needs to support aligned allocation of contiguous physical pages for a WEU across multiple planes in aligned blocks, similar to vertical and horizontal super-page striping [3]. Second, to quantify the best-case of using SSD as a cache, we push the cache replacement functionality to the FTL, since the FTL has perfect information about page state. Thus, a new interface allows the CacheManager to update indices and implement write-back mode before eviction. We experimented with multiple variants and present WEU-LRU, an update to the greedy SSD GC algorithm that replaces WEUs.

We also added the SATA TRIM command [28] in our simulator, which invalidates a range of SSD logical addresses. When the CacheManager issues TRIM commands, the SSD performs GC without copying forward data. Our SSD simulator is based on well-studied simulators [1, 3] with a hybrid mapping scheme [15] where blocks are categorized into data and log blocks. Page-mapped log blocks will be consolidated into block-mapped data blocks through merge operations. Log blocks are further segregated into sequential regions and random areas to reduce expensive merge operations.

Prototype system. We have implemented a prototype Nitro system in user space, leveraging multi-threading and asynchronous I/O to increase parallelism and with support for replaying storage traces. We use real SSDs for our cache, and either a COS or TPS system with hard drives for storage (§ 5). We confirmed the cache hit ratios are the same between the simulator and prototypes. When evicting dirty extents from SSD, they are moved to a write queue and written to disk storage before their corresponding WEU is replaced.

5 Experimental Methodology

In this section, we first describe our analysis metrics. Second, we describe several storage traces used in our experiments. Third, we discuss the range of parameters explored in our evaluation. Fourth, we present the platform for our simulator and prototype systems.

5.1 Metrics

Our results present overall system IOPS, including both reads and writes. Because writes are handled asynchronously and are protected by NVRAM, we further focus on read-hit ratio and read response time to validate Nitro. The principal evaluation metrics are:

IOPS: Input/Output operations per second.

Read-hit ratio: The ratio of read I/O requests satisfied by Nitro over total read requests.

Read response time (RRT): The average elapsed time from the dispatch of one read request to when it finishes, characterizing the user-perceivable latency.

SSD erasures: The number of SSD blocks erased, which counts against SSD lifespan.

Deduplication and compression ratios: Ratio of the data size versus the size after deduplication or compression ($\geq$ 1X). Higher values indicate more space savings.

5.2 Experimental Traces

Most of our experiments are with real-world traces, but we also use synthetic traces to study specific topics.

FIU traces: Florida International University (FIU) collected storage traces across multiple weeks, including WebVM (a VM running two web-servers), Mail (an email server with small I/Os), and Homes (a file server with a large fraction of random writes). The FIU traces contain content fingerprint information with small granularity (4KB or 512B), suitable for various extent size studies. The FIU storage systems were reasonably sized, but only a small region of the file systems was accessed during the trace period. For example, WebVM, Homes and Mail have file system sizes of 70GB, 470GB and 500GB in size, respectively, but we measured that the traces only accessed 5.3%, 5.8% and 11.5% of the storage space, respectively [14]. The traces have more writes than reads, with write-to-read ratios of 3.6, 4.2, and 4.3, respectively. To our knowledge, the FIU traces are the only publicly available traces with content.

Boot-storm trace: A "boot-storm" trace refers to many VMs booting up within a short time frame from the same storage system [8]. We first collected a trace while booting up one 18MB VM kernel in Xen hypervisor. The trace consisted of 99% read requests, 14% random I/O, and 1.2X deduplication ratio. With this template, we synthetically produced multiple VM traces in a controlled manner representing a large number of cloned VMs with light changes. Content overlap was set at 90% between VMs, and the addresses of duplicates were shifted by 0-15% of the address space.

Restore trace: To study snapshot restore, we collected 100 daily snapshots of a 38GB workstation VM with a median over-write rate of 2.3%. Large read I/Os (512KB) were issued while restoring the entire VM.

Fingerprint generation. The FIU traces only contain fingerprints for one block size (e.g. 4KB), and we want to vary the extent size for experiments (4-128KB), so it is necessary to process the traces to generate extent fingerprints. We use a multi-pass algorithm, which we briefly describe. The first pass records the fingerprints for each block read in the trace, which is the initial state of the file system. The second pass replays the trace and creates extent fingerprints. An extent fingerprint is generated by calculating a SHA-1 hash of the concatenated block fingerprints within an extent, filling in unspecified block fingerprints with unique values as necessary. Write I/Os within the trace cause an update to block fingerprints and corresponding extent fingerprints. A final pass replays the modified trace for a given experiment.

Synthetic compression information. Since the FIU traces do not have compression information, we synthetically generate content with intermingled unique and repeated data based on a compression ratio parameter. Unless noted, the compression ratio is set for each extent using a normal distribution with mean of 2 and variance of 0.25, representing a typical compression ratio for primary workloads [29]. We used LZ4 [9] for compression and decompression in the prototype.

Variable	Values
Fingerprint index ratio (%)	**100**, 75, 50, 25, 0 (off)
Compression	**on**, off
Extent size (KB)	4, **8**, 16, 32, 64, 128
Write/Evict granularity	**WEU**, extent
Cache size (% of volume)	0.5, 1, **2**, 5
WEU size (MB)	0.5, 1, **2**, 4
Co-design	**standard SSD**, modified SSD
Write-mode	write-through, **write-back**
Prefetching	dynamic up to 128KB
Backend storage	COS, TPS

Table 1: Parameters for Nitro with default values in bold.

5.3 Parameter Space

Table 1 lists the configuration space for Nitro, with default values in bold. Due to space constraints, we interleave parameter discussion with experiments in the evaluation section. While we would like to compare the impact of compression using WEU-caching versus plain extent-based caching, it is unclear how to efficiently store compressed (variable-sized) extents to SSDs without using WEUs or an equivalent structure [10, 19, 31]. For that reason, we show extent caching without compression, but with or without deduplication, depending on the experiment. The cache is sized as a fraction of the storage system size. For the FIU traces, a 2% cache corresponds to 1.4GB, 9.4GB, and 10GB for WebVM, Homes and Mail traces respectively. Most evaluations are with the standard SSD interface except for a co-design evaluation. We use the notation Deduplicated (D), Non-deduplicated (ND), Compressed (C) and Non-compressed (NC). Nitro uses the WEU (D, C) configuration by default.

5.4 Experimental Platform

Our prototype with a COS system is a server equipped with 2.33GHz Xeon CPUs (two sockets, each with two cores supporting two hardware threads). The system has 36GB of DRAM, 960MB of NVRAM, and two shelves of hard drives. One shelf has 12 1TB 7200RPM SATA hard drives, and the other shelf has 15 7200RPM 2TB drives. Each shelf has a RAID-6 configuration including two spare disks. For comparison, the TPS system is a server equipped with four 1.6GHz Xeon CPUs and 8GB DRAM with battery protection. There are 11 1TB 7200RPM disk drives in a RAID-5 configuration. Before each run, we reset the initial state of the HDD storage based on our traces.

Both prototypes use a Samsung 256GB SSD, though our experiments use a small fraction of the available SSD, as controlled by the cache capacity parameter. According to specifications, the SSD supports >100K random read IOPS and >90K random write IOPS. Using a SATA-2 controller (3.0 Gbps), we measured 8KB SSD random reads and writes at 18.7K and 4.2K IOPS, respectively. We cleared the SSD between experiments.

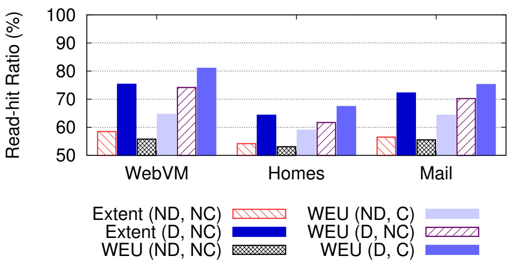


Figure 4: Read-hit ratio of WEU-based vs. Extent-based for all workloads. Y-axis starts at 50%.

We set the SSD simulation parameters based on the Micron MLC SSD specification [23]. We vary the size of each block or flash chip to control the SSD capacity. Note that a larger SSD block size has longer erase time (e.g., 2ms for 128KB and 3.8ms for 2MB). For the unmodified SSD simulation, we over-provision the SSD capacity by 7% for garbage collection, and we reserve 10% for log blocks for the hybrid mapping scheme. No space reservation is used for the modified SSD WEU variants.

6 Evaluation

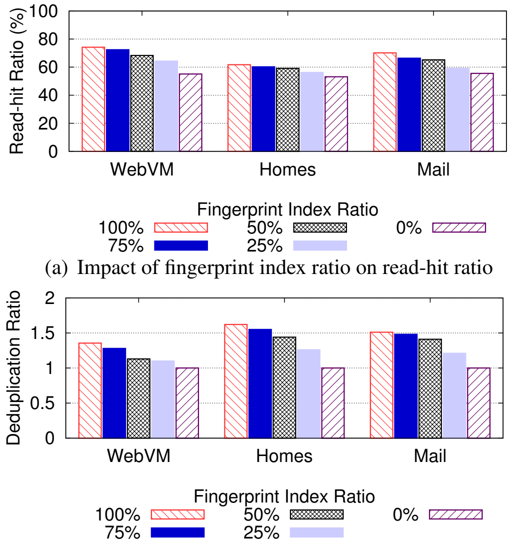
This section presents our experimental results. We first measure the impact of deduplication and compression on caching as well as techniques to reduce in-memory indices and to extend SSD lifespan. Second, we evaluate Nitro performance on both COS and TPS prototype systems and perform sensitivity and overhead analysis. Finally, we study Nitro's additional advantages.

6.1 Simulation Results

We start with simulation results, which demonstrate caching improvements with deduplication and compression and compare a standard SSD against a co-design that modifies an SSD to specifically support caching.

Read-hit ratio. We begin by showing Nitro's effectiveness at improving the read-hit ratio, which is shown in Figure 4 for all three FIU traces. The trend for all traces is that adding deduplication and compression increases the read-hit ratio.

WEU (D, C) with deduplication (fingerprint index ratio set to 100% of available SSD extent entries) and compression represents the best scenario with improvements of 25%, 14% and 20% across all FIU traces as compared to a version without deduplication or compression (WEU (ND, NC)). Adding compression increases the read-hit ratio for WEU by 5-9%, and adding deduplication increases the read-hit ratio for WEU by 8-19% and extents by 6-17%. Adding deduplication consistently offers a greater improvement than adding compression, suggesting deduplication is capable of increasing the read-hit ratio for primary workloads that contain many duplicates



(a) Impact of fingerprint index ratio on read-hit ratio

(b) Impact of fingerprint index ratio on deduplication ratio

Figure 5: Fingerprint index ratio impact on read-hit ratio and deduplication for WEU (D, NC).

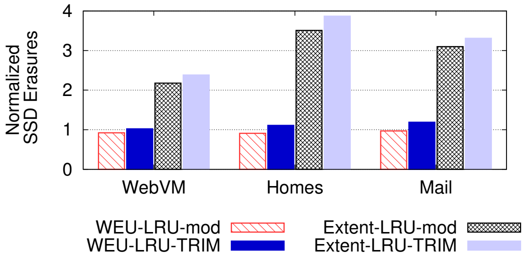


Figure 6: Number of SSD erasures for modified and unmodified SSD variants.

like the FIU traces. Comparing WEU and extent-based caching with deduplication, but without compression (D, NC), extent-based caching has a slightly higher hit-ratio by 1-4% due to finer-grained evictions. However, the advantages of extent-based caching are offset by increased SSD erasures, which are presented later. In an experiment that increased the cache size up to 5% of the file system size, the combination of deduplication and compression (D, C) showed the largest improvement. These results suggest Nitro can extend the caching benefits of SSDs to much larger disk storage systems.

Impact of fingerprint index ratio. To study the impact of deduplication, we adjust the fingerprint index ratio for WEU (D, NC). 100% means that all potential duplicates are represented in the index, while 0% means deduplication is turned off. Decreasing the fingerprint index ratio directly reduces the RAM footprint (29 bytes per entry) but also likely decreases the read-hit ratio as the deduplication ratio drops.

Figure 5(a) shows the read-hit ratio drops gradually as the fingerprint index ratio decreases. Figure 5(b) shows that the deduplication ratio also slowly decreases with the fingerprint index ratio. Homes and Mail have higher deduplication ratios (≥1.5X) than WebVM, as shown in Figure 1. Interestingly, higher deduplication ratios in the Homes and Mail traces do not directly translate to higher read-hit ratios because there are more writes than reads (∼4 W/R ratio), but do increase IOPS (§6.2). Nitro users could limit their RAM footprint by setting the fingerprint index ratio to 75% or 50%, which results in a 16-22% RAM savings respectively and a decrease in read-hit ratio of 5-11%. For example, when reducing the fingerprint index from 100% to 50% for the Mail trace (10GB cache size), ≥131,000 duplicate extents are not cached by Nitro, on average.

WEU vs. SSD co-design. So far, we considered scenarios where the SSD is unmodified. Next we compare our current design to an alternative that modifies an SSD to directly support WEU caching. In this experiment, we study the impact of silent eviction and the WEU-LRU eviction policy (discussed in §4) on SSD erasures. Our co-design specifically aligns WEUs to SSD blocks (WEU-LRU-mod). We also compare our co-design to variants using the TRIM command (WEU/extent-LRU-TRIM), which alerts the FTL that a range of logical addresses can be released. Figure 6 plots SSD erasures normalized relative to WEU-LRU without SSD modifications (1.0 on the vertical axis) and compares WEU versus extent caching.

SSD erasures are 2-4X higher for the extent-LRU-mod approach (i.e. FlashTier [26] extended to use an LRU policy) and extent-LRU-TRIM approach as compared to both WEU versions. This is because the CacheManager lacks SSD layout information so that extent-based eviction cannot completely avoid copying forward live SSD data. Interestingly, utilizing TRIM with the WEU-LRU-TRIM approach has similar results to WEU-LRU-mod, which indicates the CacheManager could issue TRIM commands before overwriting WEUs instead of modifying the SSD interface. We also analyzed the impact of eviction policy on read-hit ratio. WEU-LRU-mod achieves a 5-8% improvement in read-hit ratio compared to an unmodified version across the FIU traces.

Depending on the data set, the number of SSD erasures varied for the FTL and TRIM alternatives, with results between 9% fewer and 20% more erasures than using WEUs. So, using WEUs for caching is a strong alternative when it is impractical to modify the SSD or when the TRIM command is unsupported. Though not shown, caching extents without SSD modifications or TRIM (naive caching) resulted in several orders of magnitude more erasures than using WEUs.

6.2 Prototype System Results

Next, we report the performance of Nitro for primary workloads on both COS and TPS systems. We then

Metric (%)	***Trace***	***Extent***	***Nitro WEU Variants***			
		ND, NC	ND, NC	ND, C	D, NC	D, C
COS system						
IOPS	WebVM	251	307	393	532	661
	Homes	259	341	432	556	673
	Mail	213	254	292	320	450
RRT	WebVM	52	54	63	72	78
	Homes	46	49	55	57	62
	Mail	50	53	61	67	72
TPS system						
IOPS	WebVM	93	113	148	198	264
	Homes	90	130	175	233	287
	Mail	56	75	115	122	165
RRT	WebVM	39	41	49	58	64
	Homes	39	42	47	49	54
	Mail	41	44	51	57	61

Table 2: Performance evaluation of Nitro and its variants. We report IOPS improvement and read response time (RRT) reduction percentage relative to COS and TPS systems without an SSD cache. The standard deviation is ≤7.5%.

present sensitivity and overheads analysis of Nitro. Note that the cache size for each workload is 2% of the file system size for each dataset unless otherwise stated.

Performance in COS system. We first show how a high read-hit ratio in our Nitro prototype translates to an overall performance boost. We replayed the FIU traces at an accelerated speed to use ∼95% of the system resources, (reserving 5% for background tasks), representing a sustainable high load that Nitro can handle. We setup a warm cache scenario where we use the first 16 days to warm the cache and then measure the performance for the following 5 days.

Table 2 lists the improvement of total IOPS (reads and writes), and read response time reduction relative to a system without an SSD cache for all FIU traces. For example, a decrease in read response time from 4ms to 1ms implies a 75% reduction. For all traces, IOPS improvement is ≥254%, and the read response time reduction is ≥49% for Nitro WEU variants. In contrast, the Extent (ND, NC) column shows a baseline SSD caching system without the benefit of deduplication, compression, or WEU. The read-hit ratio is consistent with Figure 4.

We observe that with deduplication enabled (D, NC), our system achieves consistently higher IOPS compared to the compression-only version (ND, C). This is because finding duplicates in the SSD prevents expensive disk storage accesses, which have a larger impact than caching more data due to compression. Nitro (D, C) achieves the highest IOPS improvement (673%) in Homes using COS. As explained before, a high deduplication ratio indicates that duplicate writes are canceled, which contributes to the improved IOPS. For Mail, the increase of deduplication relative to compression-only version is smaller because small I/Os (29% of I/Os are ≤ the 8KB extent size) can cause more reads from disk on the write path, thus negating some of the benefits of duplicate hits in the SSDs.

Compared to extent-based caching, WEU (D, C) improves non-normalized IOPS up to 120% and reduces read response time up to 55%. Compared to WEU (ND, NC), extent-based caching decreases IOPS 13-22% and increases read response time 4-7%. This is partially because extent-based caching increases the SSD write penalty due to small SSD overwrites. From SSD random write benchmarks, we found that 2MB writes (WEU size) have ∼60% higher throughput than 8KB writes (extent size), demonstrating the value of large writes to SSD.

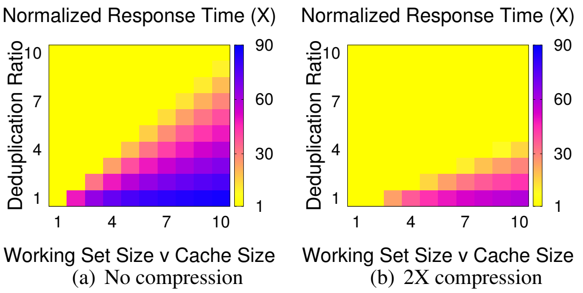
We also performed cold cache experiments that replay the trace from the last 5 days without warming up Nitro. Nitro still improves IOPS up to 520% because of sequential WEU writes to the SSD. Read response time reductions are 2-29% for Nitro variants across all traces because fewer duplicated extents are cached in the SSD.

Performance in TPS system. Nitro also can benefit a TPS system (Table 2). Note that Nitro needs to compute extent fingerprints before performing deduplication, which is computation that can be reused in COS but not TPS. In addition, Nitro cannot leverage a recipe cache for TPS to accelerate read requests, which causes 5-14% loss in read hit-ratio for our WEU variants.

For all traces, the improvement of total IOPS (reads and writes) is ≥75%, and the read response time reduction is ≥41% for Nitro WEU variants. While deduplication and compression improve performance, the improvement across Nitro variants is lower relatively than for our COS system because storage systems without capacity-optimized techniques (e.g. deduplication and compression) have shorter processing paths, thus better baseline performance. For example, overwrites in existing deduplication systems can cause performance degradation because metadata updates need to propagate changes to an entire file recipe structure. For these reasons, the absolute IOPS is higher than COS with faster read response times. Cold cache results are consistent with warm cache results.

Sensitivity analysis. To further understand the impact of deduplication and compression on caching, we use synthetic traces to investigate the impact on random read performance, which represents the worst-case scenario for Nitro. Note that adding non-duplicate writes to the traces would equivalently decrease the cache size (e.g. multi-stream random reads and non-duplicate writes). Two parameters control the synthetic traces: (1) The ratio of working set size versus the cache size and (2) the deduplication ratio. We vary both parameters from 1 to 10, representing a large range of possible scenarios.

Figure 7 shows projected 2D contour graphs from a 3D plot for (D, NC) and (D, C). The metric is read re-



(a) No compression (b) 2X compression

Figure 7: Sensitivity analysis of (D, NC) and (D, C).

sponse time in COS with Nitro normalized against that of fitting the entire data set in SSD (lower values are better). The horizontal axis is the ratio of working set size versus cache size, and the vertical axis is the deduplication ratio. The bottom left corner (1, 1) is a working set that is the same size as the cache with no deduplication. We can derive the effective cache size from the compression and deduplication ratio. For example, the effective cache size for a 16GB cache in this experiment expands to 32GB with a 2X deduplication ratio configuration, and further to 64GB when adding 2X compression.

First, both deduplication and compression are effective techniques to improve read response time. For example, when the deduplication ratio is high (e.g. ≥5X such as for multiple, similar VMs), Nitro can achieve response times close to SSD even when the working set size is 5X larger than the cache size. The combination of deduplication and compression can support an even larger working set size. Second, when the deduplication ratio is low (e.g.≤2X), performance degrades when the working set size is greater than twice the cache size. Compression has limited ability to improve response time, and only a highly deduplicated scenario (e.g. VM boot-storm) can counter a large working set situation. Third, there is a sharp transition from high response time to low response time for both (D, NC) and (D, C) configurations (values jump from 1 to > 8), which indicates that (slower) disk storage has a greater impact on response time than (faster) SSDs. As discussed before, the performance for Nitro in the TPS system is always better than the COS system.

Nitro overheads. Figure 8 illustrates the performance overheads of Nitro with low and high hit-ratios. We performed a boot-storm experiment using a real VM boot trace (§5) synthetically modified to create 60 VM traces. For the 59 synthetic versions, we set the content overlap ratio to 90%. We set the cache size to achieve 0% (0GB) and 100% (1.2GB) hit-ratios in the SSD cache. With these settings, we expect Nitro's performance to approach the performance of disk storage and SSD storage.

In both COS and TPS 0% hit-ratio configurations, we normalized against corresponding systems without SSDs. All WEU variants impose ≤7% overhead in response time because extent compression and fingerprint calculation are performed off the critical path. In the 100% hit-ratio scenario, we normalize against a system with all data fitting in SSD without WEUs. WEU (ND, NC) imposes a 2% increase in response time. Compression-only (ND, C) and deduplication-only (D, NC) impose 11% and 6.2% overhead on response time respectively. WEU (D, C) overhead (≤18%) mainly comes from decompression, which requires additional time when reading compressed extents from SSD. Although we are not focused on comparing compression algorithms, we did quantify that gzip achieves 23-47% more compression than LZ4 (our default), which improves the read-hit ratio, though decompression is 380% slower for gzip.

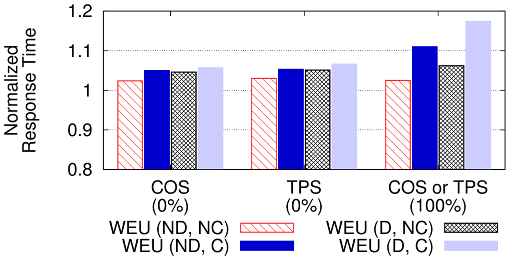


Figure 8: Overheads of Nitro prototypes with the cache sized to have 0% and 100% hit-ratios. Y-axis starts at 0.8. The standard dev. is ≤3.8% in all cases.

6.3 Nitro Advantages

There are additional benefits because Nitro effectively expands a cache: improved random read performance in aged COS, faster snapshot restore performance, and write reductions to SSD.

Random read performance in aged COS system. For HDD storage systems, unfortunately, deduplication can lead to storage fragmentation because a file's content may be scattered across the storage system. A previous study considered sequential reads from large backup files [18], while we study the primary storage case with random reads across a range of I/O sizes.

Specifically, we wrote 100 daily snapshots of a 38GB desktop VM to a standard COS system, a system augmented with the addition of a Nitro cache, and a TPS system. To age the system, we implemented a retention policy of 12 weeks to create a pattern of file writes and deletions. After writing each VM, we measured the time to perform 100 random reads for I/O sizes of 4KB to 1MB. The Nitro cache was sized to achieve a 50% hit ratio (19GB). Figure 9 shows timing results for the 1st generation (low-gen) and 100th generation (high-gen) normalized to the response time for COS low-gen at 4KB. For the TPS system, we only plot the high-gen numbers, which were similar to the low-gen results, since there was no deduplication-related fragmentation.

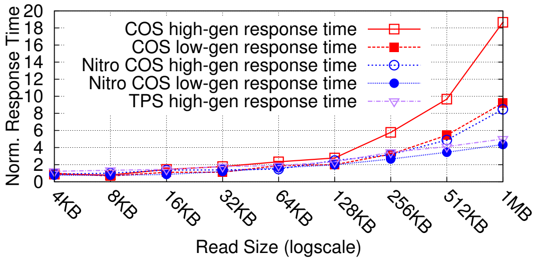


Figure 9: Response time diverges for random I/Os.

As the read size grows from 4KB to around 128KB, the response times are stable and the low-gen and high-gen results are close to each other for all systems. However, for larger read sizes in the COS high-gen system, the response time grows rapidly. The COS system's logs indicate that the number of internal I/Os for the COS system is consistent with the high response times. In comparison to the COS system, the performance gap between low-gen and high-gen is smaller for Nitro. For 1MB random reads, Nitro COS high-gen response times (76ms) are slightly faster than COS low-gen, and Nitro COS low-gen response times (39ms) are slightly faster than a TPS high-gen system. By expanding an SSD cache, Nitro can reduce performance differences across random read sizes, though the impact of generational differences is not entirely removed.

Snapshot restore. Nitro can also improve the performance of restoring and accessing standard snapshots and clones, because of shared content with a cached primary version. Figure 10 plots the restore time for 100 daily snapshots of a 38GB VM (same sequence of snapshots as the previous test). The restore trace used 512KB read I/Os, which generate random HDD I/Os in an aged, COS system described above.

We reset the cache before each snapshot restore experiment to the state when the 100th snapshot is created. We evaluate the time for restoring each snapshot version and report the average for groups of 25 snapshots with the cache sized at either 2% or 5% of the 38GB volume. The standard deviation for each group was ≤7s. Group 1-25 has the oldest snapshots, and group 76-100 has the most recent. For all cache sizes, WEU (D, C) has consistently faster restore performance than a compression-only version (ND, C). For the oldest snapshot group (1-25) with a 5% cache size, WEU (D, C) achieves a shorter restore time (374s) when deduplication and compression are enabled as compared to the system with compression only (513s). The recent snapshot group averages 80% content overlap with the primary version, while the oldest group averages 20% content overlap, as plotted against the right axis. Clearly, deduplication assists Nitro in snapshot restore performance.

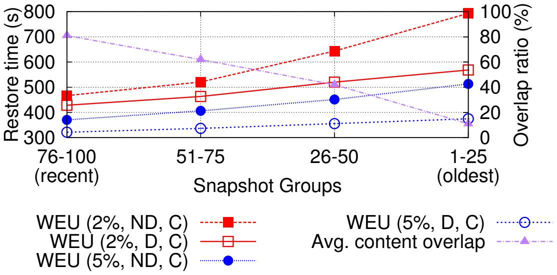


Figure 10: Nitro improves snapshot restore performance.

Reducing writes to SSD. Another important issue is how effective our techniques are at reducing SSD writes compared to an SSD cache without Nitro. SSDs do not support in-place update, so deduplication can prevent churn for repeated content to the same, or a different, address. For WebVM and Mail, deduplication-only and compression-only reduces writes to SSD (≥22%), in which compression produces more savings compared to deduplication. In the Homes, deduplication reduces writes to SSD by 39% because of shorter fingerprint reuse distance. Deduplication and compression (D, C) reduces writes by 53%. Reducing SSD writes directly translates to extending the lifespan.

7 Related Work

SSD as storage or cache. Many studies have focused on incorporating SSDs into the existing hierarchy of a storage system [2, 7, 12, 16, 30]. In particular, several works propose using flash as a cache to improve performance. For example, Intel Turbo Memory [21] adds a nonvolatile disk cache to the hierarchy of a storage system to enable fast start-up. Kgil et al. [11] splits a flash cache into separate read and write regions and uses a programmable flash memory controller to improve both performance and reliability. However, none of these systems combine deduplication and compression techniques to increase the effective capacity of an SSD cache.

Several recent papers aim to maximize the performance potential of flash devices by incorporating new strategies into the established storage I/O stack. For example, SDF [25] provides a hardware/software co-designed storage to exploit flash performance potentials. FlashTier [26] specifically redesigned SSD functionality to support caching instead of storage and introduced the idea of silent eviction. As part of Nitro, we explored possible interface changes to the SSD including aligned WEU writes and TRIM support, and we measured the impact on SSD lifespan.

Deduplication and compression in SSD. Deduplication has been applied to several primary storage systems. iDedup [27] selectively deduplicates primary workloads inline to strike a balance between performance and capacity savings. ChunkStash [6] designed a fingerprint index in flash, though the actual data resides on disk. Dedupv1 [22] improves inline deduplication by leveraging the high random read performance of SSDs. Unlike these systems, Nitro performs deduplication and compression within an SSD cache, which can enhance the performance of many primary storage systems.

Deduplication has also been studied for SSD storage. For example, CAFTL [4] is designed to achieve best-effort deduplication using an SSD FTL. Kim et al. [13] examined using the on-chip memory of an SSD to increase the deduplication ratio. Unlike these systems, Nitro performs deduplication at the logical level of file caching with off-the-shelf SSDs. Feng and Schindler [8] found that VDI and long-term CIFS workloads can be deduplicated with a small SSD cache. Nitro leverages this insight by allowing our partial fingerprint index to point to a subset of cached entries. Another distinction is that since previous deduplicated SSD projects worked with fixed-size (non-compressed) blocks, they did not have to maintain multiple references to variable-sized data. Nitro packs compressed extents into WEUs to accelerate writes and reduce fragmentation. SAR [20] studied selective caching schemes for restoring from deduplicated storage. Our technique uses recency instead of frequency for caching.

8 Conclusion

Nitro focuses on improving storage performance with a capacity-optimized SSD cache with deduplication and compression. To deduplicate our SSD cache, we present a fingerprint index that can be tuned to maintain deduplication while reducing RAM requirements. To support the variable-sized extents that result from compression, our architecture relies upon a Write-Evict Unit, which packs extents together and maximizes the cache hit-ratio while extending SSD lifespan. We analyze the impact of various design trade-offs involving cache size, fingerprint index size, RAM usage, and SSD erasures on overall performance. Extensive evaluation shows that Nitro can improve performance in both COS and TPS systems.

Acknowledgments

We thank Windsor Hsu, Stephen Manley and Darren Sawyer for their guidance on Nitro. We also thank our shepherd Vivek Pai and the reviewers for their feedback.

References

[1] N. Agrawal et al. Design Tradeoffs for SSD Performance. USENIX ATC, 2008.

[2] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. NSDI, 2011.

[3] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. ASPLOS, 2009.

[4] F. Chen, T. Luo, and X. Zhang. CAFTL: A Content-aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives. FAST, 2011.

[5] C. Constantinescu, J. Glider, and D. Chambliss. Mixing Deduplication and Compression on Active Data Sets. DCC, 2011.

[6] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. USENIX ATC, 2010.

[7] Facebook Inc. Facebook FlashCache, 2013. `https://github.com/facebook/flashcache`.

[8] J. Feng and J. Schindler. A Deduplication Study for Host-side Caches in Virtualized Data Center Environments. MSST, 2013.

[9] Google LZ4: Extremely Fast Compression Algorithm. Google, 2013. `http://code.google.com/p/lz4`.

[10] W. Huang et al. A Compression Layer for NAND Type Flash Memory Systems. ICITA, 2005.

[11] Kgil, Taeho and Roberts, David and Mudge, Trevor. Improving NAND Flash Based Disk Caches. ISCA, 2008.

[12] H. Kim and S. Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. FAST, 2008.

[13] J. Kim et al. Deduplication in SSDs: Model and Quantitative Analysis. MSST, 2012.

[14] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM TOS*, 2010.

[15] S. Lee et al. A Log Buffer-based Flash Translation Layer Using Fully-associative Sector Translation. *ACM TECS*, 2007.

[16] C. Li et al. Quantifying and Improving I/O Predictability in Virtualized Systems. IWQoS, 2013.

[17] C. Li et al. Assert(!Defined(Sequential I/O)). HotStorage, 2014.

[18] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. FAST, 2013.

[19] T. Makatos et al. Using Transparent Compression to Improve SSD-based I/O Caches. EuroSys, 2010.

[20] B. Mao et al. Read Performance Optimization for Deduplication Based Storage Systems in the Cloud. *ACM TOS*, 2014.

[21] J. Matthews et al. Intel Turbo Memory: Nonvolatile Disk Caches in the Storage Hierarchy of Mainstream Computer Systems. *ACM TOS*, 2008.

[22] D. Meister and A. Brinkmann. Dedupv1: Improving Deduplication Throughput Using Solid State Drives (SSD). MSST, 2010.

[23] Micron MLC SSD Specification, 2013. `http://www.micron.com/products/nand-flash/`.

[24] D. Narayanan et al. Migrating Server Storage to SSDs: Analysis of Tradeoffs. EuroSys, 2009.

[25] J. Ouyang et al. SDF: Software-defined Flash for Web-scale Internet Storage Systems. ASPLOS, 2014.

[26] M. Saxena, M. M. Swift, and Y. Zhang. FlashTier: A Lightweight, Consistent and Durable Storage Cache. EuroSys, 2012.

[27] K. Srinivasan et al. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. FAST, 2012.

[28] TRIM Specification. ATA/ATAPI Command Set- 2 (ACS-2). `http://www.t13.org`, 2007.

[29] G. Wallace et al. Characteristics of Backup Workloads in Production Systems. FAST, 2012.

[30] Q. Yang and J. Ren. I-CASH: Intelligently Coupled Array of SSD and HDD. HPCA, 2011.

[31] K. S. Yim, H. Bahn, and K. Koh. A Flash Compression Layer for Smartmedia Card Systems. *IEEE TOCE*, 2004.